AFRL-RI-RS-TR-2013-055

OPEN COMPONENT PORTABILITY INFRASTRUCTURE (OPENCPI)

MERCURY FEDERAL SYSTEMS, INC.

MARCH 2013

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■ UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MARCH 2013	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2010 – OCT 2012

4. TITLE AND SUBTITLE OPEN COMPONENT PORTABILITY INFRASTRUCTURE (OPENCPI)	**5a. CONTRACT NUMBER** FA8750-10-C-0194
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) James Kulp, Shepard Siegel, John Miller	**5d. PROJECT NUMBER** S3MF
	5e. TASK NUMBER OC
	5f. WORK UNIT NUMBER PI
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Mercury Federal Systems, Inc. 1901 South Bell Street, Suite 402 Arlington, VA 22201	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2013-055

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2013-1075
Date Cleared:5 March 2013

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The technical advantages of utilizing advanced computer processing components to meet real-time system requirements are often offset by higher initial system development costs. This is further exacerbated because once a system is developed, that system's codes, tools and components are generally unique to that system, but must be redeveloped for a different system configuration. The Open Component Portability Infrastructure (OpenCPI) is a previously developed open source runtime framework for component-based, heterogeneous embedded computing that simplified the programming of heterogeneous processing environments. Here the OpenCPI was extended to facilitate the integration of different configurations of General Purpose Processors (GPPs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) in a multi-processing computing platform. This facilitates the advantageous exploitation of each of these processor types in a complete system that can be tailored for a specific use. The key functional gaps were filled that inhibited OpenCPI's adoption, and the technology was hardened and matured to increase the Technology Readiness Level (TRL). The "use/adoption/experimentation/trial" necessary to utilize OpenCPI was simplified, and through its implementation, the advantageous transitions of these advanced computer processing technologies for wider exploitation, particularly in the DoD community, are enabled.

15. SUBJECT TERMS
Heterogeneous Embedded Computing, FPGA, GPU, real-time systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**GEORGE RAMSEYER**
U	U	U	UU	62	**19b. TELEPONE NUMBER (*Include area code*)** **N/A**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

The transition of commercial and proprietary technology to an open source capability was previously supported by the initial development [1] of the Open Component Portability Infrastructure (OpenCPI), and that became the basis for this effort. The project described in this report built upon that OpenCPI foundation, and in so doing establishes OpenCPI as an important and viable software development framework.

This effort focused on the concepts of the software Component-Based Development (CBD). Software applications can be assembled from snippets of codes that were originally written, tested and implemented in other applications, which is expected to reduce the development costs of new acquisitions. Heterogeneous processing allows the mixing and switching of different types of processing technologies. The integration of different configurations of General Purpose Processors (GPPs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) in a multi-processing computing platform facilitates the advantageous exploitation of each of these processor types in a complete system that can be tailored for a specific use. The attributes of embedded processors make these processors necessary to meet the requirements in many military acquisitions. This is reflected in not only the additional processing power, but also in the Size, Weight and (electrical) Power (SWAP) requirements. Proprietary restrictions and other considerations have resulted in the redevelopment of legacy codes. To be fully available to the acquisition community, open source licensing is advantageous for all aspects of this technology, including the codes. Appropriate restrictions are required for sensitive technologies.

OpenCPI was developed to meet the challenges of effectively, productively, and efficiently exploiting heterogeneous processing for embedded missions. OpenCPI employs a component-based development methodology with supporting tools and a runtime environment for applications consisting of components that can be executed on several different computing processor types. OpenCPI effectively crosses the boundaries between these diverse processor technologies. Documents describing OpenCPI can be found on the OpenCPI web site [2], which includes a preliminary technical summary document [3].

This effort developed component based systems, including well-defined, open, and portable application programming environments for embedded, heterogeneous, and multi-processor applications. Also developed were application structures, the connectivity between different computing processor types, and hardware platform elements. A control plane model was developed that includes a Human/Computer Interface (HCI). A management model was developed for the install/upgrade/uninstall and the startup/shutdown of the application and hardware subsystems. The specific baseline software platform that was developed has capabilities in several areas critical to the embedded Department of Defense (DoD) needs, including communications management that is suitable for fabric-based interconnection technologies.

This project addressed the key functional gaps in the original open source prototype that have inhibited OpenCPI adoption. The technology was hardened and matured to increase the Technology Readiness Level (TRL). The barriers to its development, testing, adoption and implementation were addressed, and the suitability for applying OpenCPI to specific applications was determined.

The work proceeded according to technical, staffing, partnering and funding dependencies, but is described here in a logical technical breakdown, rather than following the actual timeline. Since OpenCPI is based on and embraces the principles of open source software, the software, firmware, FPGA codes, tools, and documentation were all posted on the *Opencpi.org* web site [2] and released under an open source license[1]. The key technical results of this work were embodied in the new and improved evolution of OpenCPI, as it addressed a combination of technical gap filling, hardening and ease-of-use enhancements.

In addition to these technical enhancements, several integration and application experiments were performed to assess and demonstrate how OpenCPI could coexist with other relevant frameworks for image and software radio processing, and how well OpenCPI served as an application environment for specific applications. These efforts included instrumentation and measurements.

Outreach and community building exposed OpenCPI to additional users inside and outside of the defense and intelligence communities. Some of this was based on *ad hoc* meetings and briefings, but there were several formal briefings of OpenCPI at conferences.

[1]Lesser GNU Public License 3 (LGPL3)

2.0 INTRODUCTION

This project consisted of two phases over 25 months that spanned from September 2010 through October 2012. This project is described as one project, since the second phase of the effort was an extension of the first phase, and was a six-month application technology transition facilitator study.

The underlying OpenCPI baseline architecture was a software-defined platform, originally motivated by the Department of Defense (DoD) Programmable Communication Terminals. It was based on and was compliant with the Software Communication Architecture (SCA) developed in the Joint Tactical Radio System (JTRS) program [4].

This report describes improvements to the capabilities and applicability of OpenCPI, and to the expansion of the actual community of users. OpenCPI addresses several dimensions of diversity and heterogeneity in embedded systems, and this project addressed each dimension in turn by expanding the range of the supported configurations.

In extending OpenCPI there was a risk that the extension would not be completely compatible with the existing software architecture. The internal OpenCPI architecture was adjusted to encompass these new extensions. A brief introduction to each effort is included here, with a more detailed description in later sections.

2.1 Authoring Model

An Authoring Model is based on a common software language and tool to provide the translation of a common software language to other processor-type specific languages. An Authoring Model also addresses abstract conceptual executions and communications. Here the communications are further complicated by the requirement that some component interfaces have multiple language translations, as different components may each have their own specific software language. Adding to this complexity is that the Authoring Model must handle diversity, as there may be more than one way to correctly program components that achieves the same functionality.

The Resource-Constrained C-language (RCC) model is a basic and simple model for writing components in the C language. At a conceptual level, this same model could be rendered and supported in other programming languages including C^{++}, Java, and Python. Here the language coverage of the RCC model was not expanded beyond C, but was simplified and expanded in several ways based on the experience that was gained from our writing new components and applications.

The Hardware Description Language (HDL) is an Authoring Model which supports the writing of codes for FPGA and Application-Specific Integrated Circuit (ASIC) processor components. As used here, the HDL was applied to the Verilog, the Very High Speed Integrated Circuit (VHSIC)-Hardware Description Language (VHDL), the System-Verilog, the Bluespec System-Verilog, and the System-C processor-specific languages. At the start of this project OpenCPI supported the Verilog language for the OpenCPI HDL Authoring Model. A major effort here led to supporting the VHDL language, since an application was received that was written in VHDL. This is an example of adding language support to an existing Authoring Model.

A new Authoring Model was designed and implemented for components executing on GPUs, which extended the OpenCPI's processor heterogeneity. This effort targeted the Open Computing Language (OpenCL) dialect of C, which is supported by most graphic and some multi-core processor vendors, and is called the OpenCL Authoring Model (OCL). It can be conceptually applied to other GPU-oriented languages and environments such as NVIDIA's Compute Unified Device Architecture (CUDA[2]). Efforts

[2] CUDA is a parallel computing platform and programming model created by NVIDIA, and implemented on NVIDIA GPUs.

for a GPU Authoring Model combined conceptualizing a new Authoring Model with specifically prototyping it in the OpenCL language dialect of C using the associated low level Application Programming Interfaces (APIs).

In summary, the RCC Authoring Model was enhanced to simplify specific use cases, and the HDL model [5] was extended with support for the VHDL language. A new GPU-oriented Authoring Model was defined, and OpenCL was the target language.

2.2 Data Plane

OpenCPI software that supports the data plane mechanism is a transfer driver. As defined in the OpenCPI Technical Summary [3], a data plane is the means by which system components communicate with each other at runtime. The different components and parts of real heterogeneous embedded systems use several mechanisms to connect with each other. The top-level distinction among such communication methods is whether the communications are collocated. It is advantageous for collocated communications to occur among processor components in the same execution environment. To achieve this, an Authoring Model must maximize the efficiency of collocated communications.

This project developed support for additional key data plane scenarios that had not been previously supported. A new data plane mechanism was implemented for clustered applications, which targeted the remote Direct Memory Access (DMA) mechanisms and APIs promulgated by the Open Fabrics Enterprise Distribution[3] (OFED), which is typically deployed over either 10 Gbps or 40 Gbps Infiniband hardware in clustered environments. As with all of the OpenCPI transfer drivers, this is transparent to how the components are written in the Authoring Models. This process allowed for collocated components that were executed on a single processor, or on separate processors comprising a cluster connected with Infiniband, to execute without being changed or recompiled.

A datagram is a self-contained, independent entity of data that carries sufficient information to be routed from its source to a destination processor without the reliance of earlier exchanges between its source, the destination processor and the transporting network. Support was developed for datagram transports by the OpenCPI data plane, which additionally guaranteed message delivery and the order of the message delivery. An application-level communication system to support these datagram transports is the Network File System (NFS). Similarly, the Remote Direct Memory Access (RDMA) model of communication [6] used by the OpenCPI data plane can benefit from directly using datagram transports such as User Datagram Protocol (UDP)/Internet Protocol (IP), as well as the link layer (L2) Ethernet Protocol. Support for an FPGA platform by Ethernet transport was also developed.

Another class of the OpenCPI data plane communication is components communicating to the outside world. There are scenarios where applications require the inputs or outputs of a component's port to communicate outside of the OpenCPI environment, and not to another OpenCPI component. A different driver type provides a bridge between the OpenCPI data plane system and the external communication/middleware systems. Two such external connections were developed that allowed applications to connect a component port to Data Distribution Service[4] (DDS) and to the Common Object Request Broker Architecture[5] (CORBA) middleware. These bridges allowed the OpenCPI applications to communicate with other middleware systems without changing a line of code or recompiling components.

[3] http://www.openfabrics.org.

[4] publish-subscribe data to other DDS applications

[5] Send and receive messages are communicated to external applications using CORBA.

2.3 Platforms

Embedded systems are built with a wide range of technologies. The OpenCPI model of a platform includes the processors in a system, as well as the ways that the processors are interconnected. Each class or type of processor, operating system, and compiler tool set is specifically supported. The processor interconnection technologies are supported for the data-plane communications, and some of the communication paths are also supported for the control-plane communications. There were two major new platform efforts in this project, the support for the Altera FPGA platforms and associated tools, and specific support for the application-oriented platform Ettus N210 Universal Software Radio Peripheral.

The Altera efforts advanced the support for the FPGA vendor Altera to bring it to a level similar of that of the existing OpenCPI support for the FPGA vendor Xilinx. This included both device and hardware level support, including the PCI bus/fabric connections, the memory controllers, and tool support. The inexpensive tools that were available were specific to each of these two vendors.

The Ettus N210 platform efforts were a combination of supporting a new FPGA processing chip, the Xilinx Spartan 3A-DSP3400, and associated tools. Also supported were the Ethernet interconnect for the control and the data planes, and the FPGA implementation of the datagram protocol. The Input-Output (IO) devices required by the application were also supported, including the attached 2-channel quadrature Analog-to-Digital Converter (ADC).

The basic generic support for the FPGA platforms was also enhanced to simultaneously discover and support the multiple platforms connected either via PCI Express fabrics or link layer Ethernet connections. This also included a Universal Unique Identifier (UUID) mechanism for determining whether the currently loaded FPGA bitstream was the one needed by an application, which was reloaded as needed.

2.4 Integration

Even though mixed application-specific frameworks frequently embody lower level middleware and execution models that are incompatible with each other, these frameworks are commonly used in embedded systems. OpenCPI is non-application-specific and can be used across a range of application areas. Several experimental tasks were undertaken to combine OpenCPI with other frameworks to determine when coexistence synergies were possible.

The widely used open source computer vision library Open Computer Vision (OpenCV) was experimentally integrated with OpenCPI in collaboration [7] with the Massachusetts Institute of Technology. This OpenCV application was embedded inside of OpenCPI, and then ported. OpenCPI image-processing components were experimentally determined to directly reproduce those of OpenCV. This not only showed coexistence, but also demonstrated that the transfer of an algorithm code from one open source project to another was possible. This new set of image processing components was then added to the OpenCPI component library.

The Software Communication Architecture (SCA) is a DoD-created middleware framework for software radio applications. It was an important motivator for the early OpenCPI work, and established the value of having component-based designs in embedded DoD applications. When OpenCPI was created in an earlier project, it was developed as a stand-alone capability that did not rely on other middleware. In this project a preliminary integration was performed to prove that a SCA-compliant system could be created using OpenCPI as the underlying component, which avoided any inefficiencies of completely layering one application on top of the other. Since the SCA did not have a heterogeneous component model, non-GPP processors were supported in this integration by creating high-level proxy CORBA objects for the underlying OpenCPI components. The SCA requires all of the components to be represented as CORBA objects.

2.5 Ease of Use

Efforts were undertaken to simplify the user experience, and other efforts focused on first-time users. Earlier efforts focused on the process of creating components. Here several significant features were added to simplify and clarify how to describe an application comprised of controlled and executed components. These included an updated and simplified API for the main program to create, connect, and control components in an application, a library path mechanism to enable components in an application to be found in a number of different component libraries, and mechanisms to allow convenient textual component property values to be applied to components, including sequences, structures and multidimensional arrays.

Another effort focused on enabling entire applications to be described in the Extensible Markup Language (XML) and executed by a utility program, which entirely avoided writing or compiling top level applications for the main program. These included the XML attributes that determine when an application was completed, and an expression-based scoring system for selecting among the alternative feasible component implementations found by the library search path mechanism.

2.6 Maturity/Testing

Increasing the maturity and robustness of OpenCPI was an important goal of this project. Early in the project a number of unit test and continuous integration system tools were evaluated, and the *gtest*[6] and the Hudson[7] [8] were selected. Both of these tools were installed and configured, which resulted in additional testing capabilities in the OpenCPI repository.

A major improvement in the robustness of OpenCPI was achieved as new components, examples, and applications were developed and tested, and this resulted in updated OpenCPI software. Several FPGA and software component unit testing capabilities were also developed that included FPGA elements for inter-component protocol monitoring, and generic unit testing for both software and FPGA implementations utilizing the same component.

2.7 Performance Measurements

Performance measurements were necessary so that performances could be determined during component and application development. Platform/infrastructure developers and system engineers need to understand the system level performance characteristics in order to achieve the correct and appropriate performance evaluations. General measurements are necessary when evaluating the OpenCPI framework for use in other projects.

At the start of the project, OpenCPI had some timekeeping and event recording capabilities. The timestamping of some events inside FPGA platforms, while synchronized with GPS time, was not synchronized with the software event timekeeping subsystem. These time-bases were enhanced to provide synchronized time across the FPGAs and other software processors in a system. Events that were recorded based on different system clocks were correlated in output mechanisms utilizing a GTKWave[8] waveform viewer and an Excel spreadsheet. The granularity and accuracy of software-timed events was also improved.

---

[6] Google's C++ unit testing framework

[7] since renamed Jenkins, a continuous integration and build server

[8] GTKWave is a fully featured GTK+ based waveform viewer which reads Fast Simulator Trace (FST), interLaced eXtensible Trace (LXT), LXT2, Verilog/VHDL Zipped Trace (VZT), and GHW (format generated by the open source VHDL simulator GHDL) files and standard Verilog VCD/EVCD (format specified in IEEE-1364 and generated natively by Verilog simulators) files, and allowed their viewing.

To capture events during component executions, new event capture instrumentation points were inserted into the runtime infrastructure. As part of the protocol monitoring enhancements for testing FPGA components, the capability was added to capture and timestamp Input-Output (I-O) events at each of the component ports. To fully exploit the downstream display and analysis tools, software was developed to convert the event data to standard formats. These standard formats included the Value Change Dump (VCD) for hardware simulation viewers, and the Comma Separated Value (CSV) files formatted for import into the Excel and other tools. This provided additional comprehensive measurement capabilities for the component, the application, and the infrastructure elements.

2.8 Applications

Several OpenCV applications were re-implemented under OpenCPI as part of the integration effort. These not only demonstrated the OpenCPI suitability for such image processing applications, but also provided examples and test applications for other features, including processor allocation capabilities. Related to the application-level ease of use efforts that were previously mentioned, a load-balancing capability was created that allowed an application's software components to be deployed to more or fewer processors. This affects the trade-off between improved performances and the sharing of limited computing resources within a computing system with other applications.

Preparations were made for the implementation and porting of a Signals Intelligence (SIGINT) application. Several front-end components of the applications pre-existed, including the Digital Signal Processor (DSP) functions implemented inside the Ettus N210 platform. This work defined the heterogeneous OpenCPI components that would implement the same functionality[9] in FPGAs, GPPs, and GPUs. Several of these were implemented in software that then performed the functions of the FPGA versions. Before this task was completed, the SIGINT application was determined to be unavailable, and focus shifted to a Frequency Shift Keying (FSK) radio application, which reused a few of the components that were created for the SIGINT application. The second application required an additional major effort in VHDL support for the HDL Authoring Model.

2.9 Community

To build a community of users, two conference workshops were presented and conducted with OpenCPI. The first conference was the Wireless/Software Defined Radio (SDR) Innovation Forum on November 30, 2010 at Arlington VA, and the other was the 19th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) in May 2011 at Salt Lake City, UT. Both conferences were well attended, and the workshops were well received.

A community and interested-party meeting was held in January 2012, and was hosted by the MITRE Corporation in McLean, VA. This meeting was attended by a mixture of academic, government and industry representatives. Originally it was planned to also participate in the NVIDIA GPU Technology Conference in October 2011, but that conference was postponed by six months, and caused scheduling difficulties.

Another effort to enhance the community was a major upgrade to the OpenCPI web site [2], which provided ease of navigation to the documentation and source code repository, and included access to the bug-tracking subsidiary site.

[9] e.g. Digital Down-Converters (DDCs), complex mixers

3.0 METHODS AND PROCEDURES

3.1 Authoring Models for Different Processing Technologies

OpenCPI Authoring Models (OCLs) represented different ways to write worker[10] using languages and APIs that allowed the effective use of a class of computing technologies. For some models, a specific programming language was implied[11], whereas for others[12], the model was separate from and independent of the actual language used. Currently there is no single model[13] that exploits all of the processing technologies, although the Authoring Models must coexist and interoperate. Thus, the Authoring Models were developed to be interchangeable in an application without having to make top-level application changes, and without having to make changes to other application components.

Each model fits a pattern in which four types of interactions are defined, as shown in Figure 1. These interactions are the Management Interface, the Inputs, the Outputs, and the Local Services. The HDL Authoring Model was specifically written for FPGAs, and was written in the Verilog, the BlueSpec, the VHDL, and the System-C languages. The RCC Authoring Model [9] was written for C-based nodes that included embedded Central Processing Units (CPUs), microcontrollers, and DSP environments.

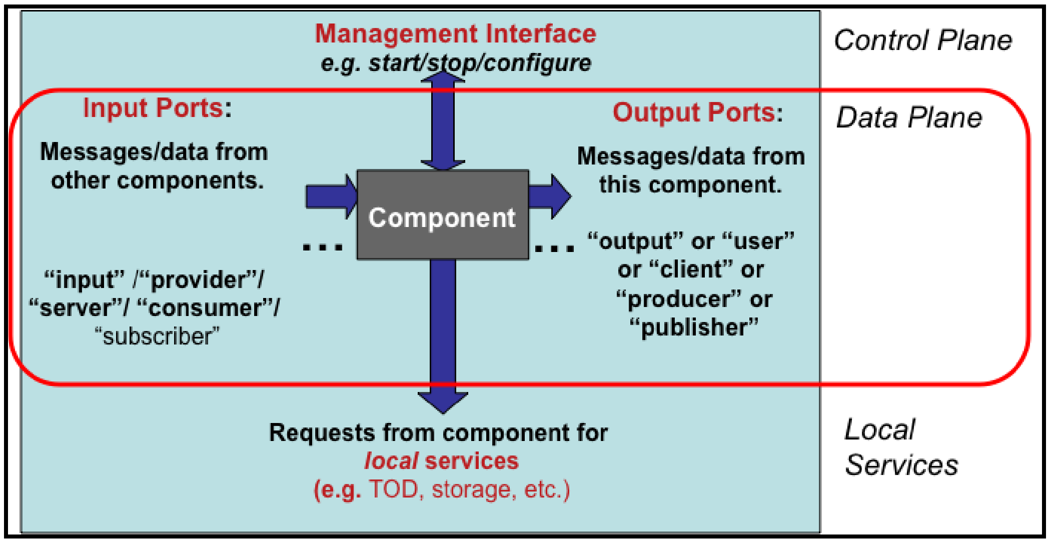

Management Interface
e.g. start/stop/configure
Control Plane
Input Ports:
Messages/data from other components.
"input" /"provider"/ "server"/ "consumer"/ "subscriber"
Component
Output Ports:
Messages/data from this component.
"output" or "user" or "client" or "producer" or "publisher"
Data Plane
Requests from component for local services (e.g. TOD, storage, etc.)
Local Services

Figure 1. Authoring Model Pattern

3.1.1 New Authoring Model for GPUs.

A new OCL was defined and implemented for a rapidly emerging and adopted class of highly parallel processors that were originally developed for graphics rendering in workstations. Today GPUs are commonly being used for non-graphical tasks. The GPUs are called General Purpose GPUs (GPGPUs)

[10] a component implementation

[11] e.g. Resource-Constrained C-language (RCC)

[12] e.g. Hardware Description Language (HDL)

[13] language/Application Programming Interface (API)

when they are used for non-graphics applications. In this project a GPGPU Authoring Model was created to exploit this class of processors.

The first challenge was to confirm that such an Authoring Model was feasible in terms of the interactions with the other existing OCLs. The second challenge was to select a language and an API that was useable across the available GPUs. The languages that the GPUs are programmed in, the GPU's APIs, the GPU's drivers, and the GPU's hardware-level details are vendor-specific and proprietary.

An abstract Authoring Model would require that each of the proprietary APIs, drivers, and hardware have their own specific model. Each abstract Authoring Model would then be defined and implementable by only one particular vendor's set of toolkits. This would result in a situation where heterogeneous requirements would be difficult to implement, document, maintain, and use; and also would be non-portable between components from different vendors.

Apple and others in the Khronos Group[14], which had standardized the Open Graphics Library (OpenGL) APIs, defined a new language called C subset and a common API. For the first time programmers were enabled to write with a single language and API, and execute the resultant code on different vendor GPUs. This new standard was OpenCL. When this was completed in 2009, the early implementations on GPUs were weak, incomplete, and did not fully exploit the computing capabilities of the GPUs. This situation improved during 2011, and further in 2012, so that the promise of write once and then execute on different GPUs is now close to reality. In general the OpenCL-based codes can move between vendors, although further hand tuning is usually required to achieve the best performance. Table 1 lists the vendors which support OpenCL on their GPUs and multi-core processors.

Table 1. OpenCL Implementations

Vendor	Hardware	Link
NVIDIA	All recent NVidia GPUs	www.nvidia.com/object/cuda_the Opencl_new.html
AMD	AMD GPU cores & x86 cores	developer.amd.com/gpu/atistreamsdk/pages/default.aspx
IBM	IBM Cell/Power architectures	www.alphaworks.ibm.com/tech/the Opencl
Intel	SSE 4.1+	http://software.intel.com/en-us/articles/intel-the Opencl-sdk/

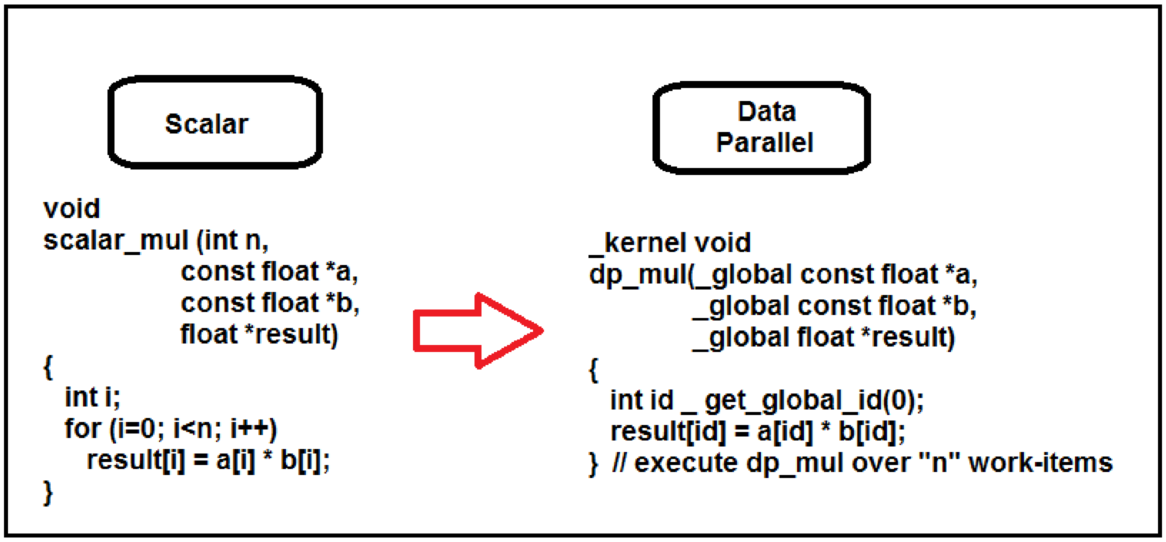
The OpenCPI model combined the best of both, with OpenCL providing a programming methodology for parallel computations that are portable across different GPU and CPU architectures, and OpenCPI supporting the replaceable component models across the processor types.

[14] http://www.khronos.org/opencl/

The key attribute of OpenCL is that it was derived as a subset/superset from International Organization for Standards (ISO) C99[15]. Recursion, function pointers, and functions in the C99 standard headers are not supported. The preprocessing directives defined by C99, and some additional built-in data types including images and vectors, are supported. Also, the geometric functions are supported that access elements relative to the context.

The context of an OpenCL function is referred to as a kernel or a work item. An OpenCL application is performed by partitioning the processing to be done among different processors. The number of work items is determined by selecting dimensional indices. There is a physical limit on the maximum number of work items. These work items are then executed in parallel. A typical example is that the workspace is an output image, and each work item is the work necessary to compute some portion[16] of that image.

OpenCL is composed of a C99-based dialect[17] and the APIs for writing and executing data/task parallel compute kernels. OpenCL focuses on its portability by trading off its convenient usability. Our two code examples in Figure 2 illustrate the difference between a simple C function and the analogous OpenCL kernel.



```
void
scalar_mul (int n,
            const float *a,
            const float *b,
            float *result)
{
  int i;
  for (i=0; i<n; i++)
    result[i] = a[i] * b[i];
}
```

```
_kernel void
dp_mul(_global const float *a,
       _global const float *b,
       _global float *result)
{
  int id _ get_global_id(0);
  result[id] = a[id] * b[id];
} // execute dp_mul over "n" work-items
```

Figure 2. C Function vs. OpenCL Kernel

These existing example OpenCL GPU kernels were reusable in OpenCPI with a simple wrapper that did not adversely affect performance. While OpenCL also included a host CPU API[18] for staging the OpenCL kernel on a GPU, the OpenCPI Component Author using the OCL did not use these APIs, but rather

[15] C99 is an informal name for *ISO/IEC 9899:1999*, a past version of the C programming language standard. It extended the previous version C90 with new linguistic and library features.

[16] tile, pixels

[17] A dialect of C that was adapted as an ANSI standard in 2000.

[18] executing, synchronizing, and data passing

executed in the background by the OpenCPI infrastructure code. This was simpler to program, as the complexity to implement OpenCL was greater than the OCL Component Author. Additionally, OpenCPI maintained commonality across the heterogeneous technologies. OpenCPI thus achieved interoperability between the different types of components communicating with each other and portability as the same code executed on different GPUs/multicores. There were common development flows of the components, of the component-based applications, and of the control applications. There was a common description for the alternative implementations including FPGAs and GPUs. There was also interchangeability, as for example GPU processing could be substituted for with FPGA processing.

The OCL model supports the kernel built-in functions in OpenCL for certain mathematical and synchronization functions, does not require or expose the OpenCL host API, and limits the required OpenCL knowledge. Host code is not required, as only device-side kernel functions need to be written. Figure 3 presents a comparison of layering of OCL vs. OpenCPI for a generic application. The OpenCPI container library handles all of the host functionality, including the loading of the binary code into the device and the allocation of memory to contain port buffers and property data. Also, the container library determines when to execute device-side kernel code based on state transitions and buffer availability.

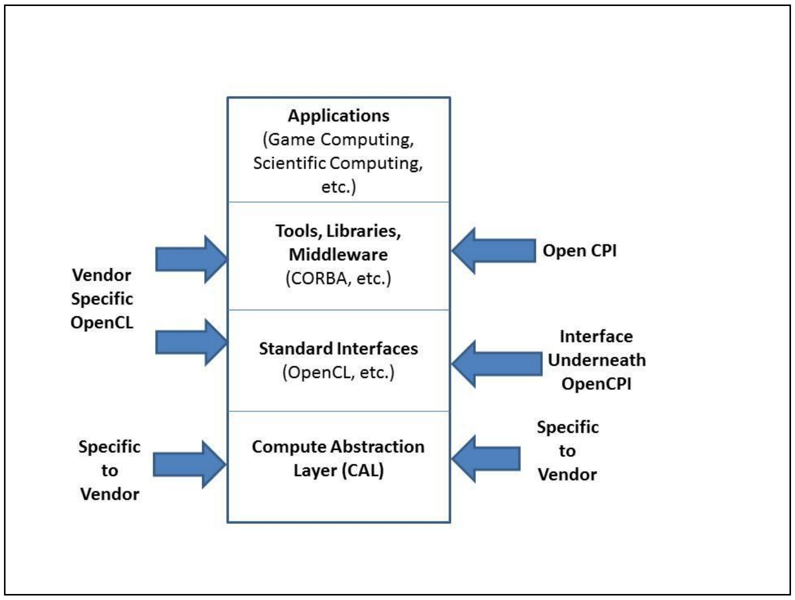

Applications
(Game Computing, Scientific Computing, etc.)
Tools, Libraries, Middleware
(CORBA, etc.)
Standard Interfaces
(OpenCL, etc.)
Compute Abstraction Layer (CAL)
Vendor Specific OpenCL
Specific to Vendor
Open CPI
Interface Underneath OpenCPI
Specific to Vendor

Figure 3. OpenCL vs. OpenCPI Layering

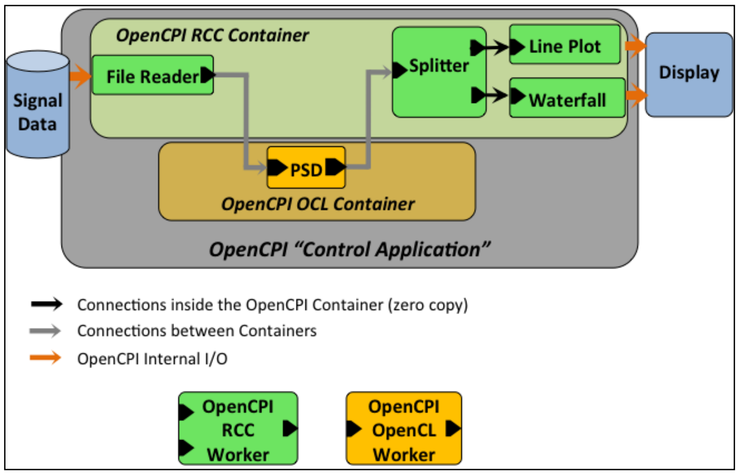


Figure 4. OCL Worker in an OpenCPI Application

The container for the OCL workers acts like other containers [10]. The control application that executes, configures and connects the workers is generic, and not specific to the OCL. The Authoring Model is completely described in the OCL Model Reference document [11]. Implementing the OCL involved enhancing two development tools, as well as the runtime code. The existing *ocpigen* tool [12], which was responsible for the code generation in all of the Authoring Models, was augmented to do the code generation. This included generating custom property structures based on all of the defined property data types and names, as well as generating a custom worker context structure that was based on metadata. The worker structure included the structures for all of the defined OpenCPI ports and the requested memory regions. The skeleton generation included stub functions for all of the entry points. Figure 4 shows an OCL worker executing inside an OCL container in an OpenCPI application that also includes other Resource-Constrained C-language (RCC) components executing in the RCC container.

3.1.2 VHDL Support for Existing HDL/FPGA Authoring Models.

The existing Hardware Description Language (HDL) Authoring Model was well-defined by the language-independent Open Core Protocol-International Partnership (OCP-IP), whose interface specifications mapped naturally to other HDLs, including Verilog, VHDL, System Verilog, Bluespec System Verilog (BSV), and System-C. Previously OpenCPI HDL partially supported the generation of Verilog code in cases when the external interface[19] of a HDL worker was rendered in Verilog.

[19] sometimes called the black box module declaration

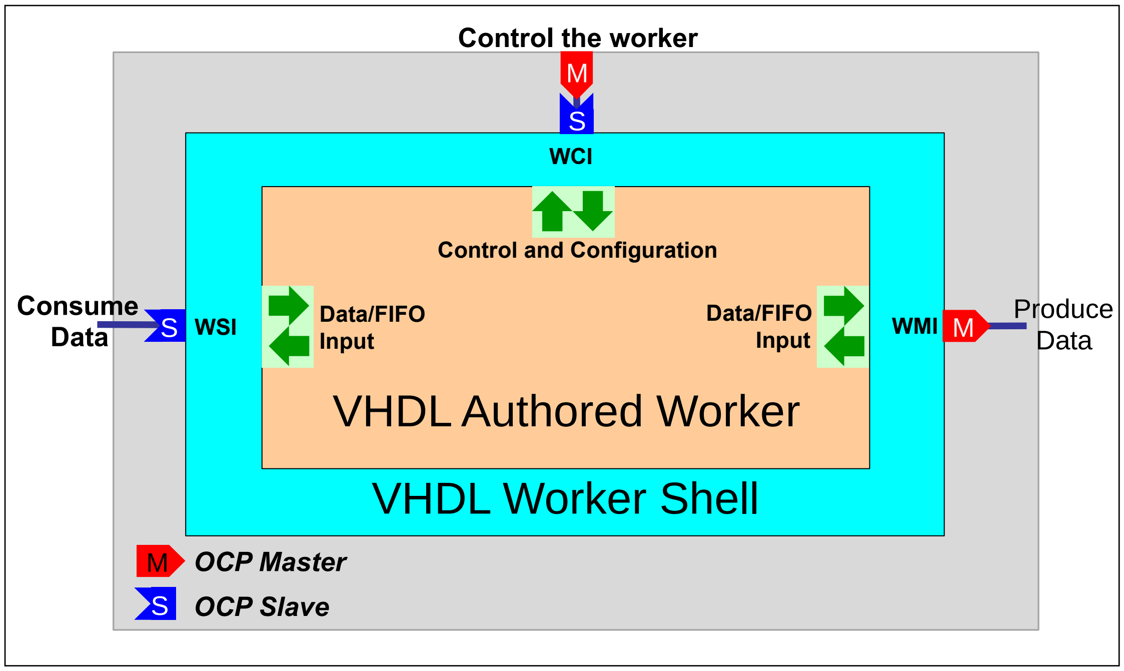


Figure 5. VHDL Worker Shell

in OpenCPI needed to go beyond generating external module definitions[20], but also needed to generate a shell that significantly simplified authoring a worker. This was especially true when wrapping existing IP Blocks to execute under OpenCPI [13]. The shell significantly reduced the complexity by ensuring that all of the exterior interoperability interfaces met the OCP-IP defined signal requirements, and provided a more worker-specific and simpler internal interface. This VHDL shell concept is described in Figure 5.

Currently Verilog supports the creation of a module declaration for an actual deployable worker, as shown on the outside of the blue layer that is represented by the red and blue arrows. VHDL also generated code for the blue shell code, as well as the entity declaration of the authored worker that must be implemented. If there is existing legacy VHDL entity architecture, then it is wrapped or modified to properly implement the generated inner entity represented by the gold inner box. Otherwise code is written to fill the gold box.

The building scripts for writing the VHDL workers in the OpenCPI component library synthesized the shell and the inner worker into a deployable synthesized module. Since this type of code generation significantly simplifies the job of implementing an OpenCPI worker, Verilog support in OpenCPI was similarly enhanced. For example, the green interfaces were much simpler than the red/blue interfaces, since the green interfaces were worker-specific.

3.1.3 Enhancements of Existing RCC Authoring Model.

An Authoring Models simplifies code writing by reducing the needed Source Lines of Code (SLOC). This is accomplished in two different ways. Codes generated by tools are developed for the worker and for the container. Also, the descriptive metadata is separated from the programming language, which results in a runtime-version of the same metadata information being made available to the infrastructure’s

[20] i.e. the entity in VHDL

containers. This results in an implicit delegation of the code from the worker to the container. A simple example of this is the RCC execution condition, where the determination of when a worker executed is not made in the worker with the typical blocking or timeout APIs, but rather in the container based on the simple run-condition data structure.

As a result of the preparation of the test, example, and application components, three enhancements were implemented that reduced the RCC worker's complexity and the SLOC. The first was for the RCC worker to indicate when the buffers had been consumed and produced, and that the worker had completed all processing. This was indicated by returning the value *RCC_ADVANCE_DONE* from the execute function of the worker. Adding this fairly simple capability allowed a typical worker to avoid maintaining an extra state while it was still processing data. This reduced the lines of code and the amount of processing required, since it eliminated a final context switch in the lifecycle of the worker.

A second enhancement was to provide a container callback function that allowed the worker to build an error reporting string using a sprint-style function, where the container took responsibility for error string storage. This allowed simple workers that reported error conditions to avoid storage management for these strings, and in most cases avoided the release method entirely.

The final enhancement was to clarify and simply the handling of dynamic execution conditions. A worker could change the execute condition to the *NULL* execute condition, which restored the initial default implied execute conditions. This allowed the workers with dynamically controlled execution conditions to easily use the default rather than having to recreate it, which further reduced the lines of code.

3.2 Data Transport Technology

Many real embedded systems have more than one processing element. A data path between the processing elements is required to communicate data and messaging. The diversity of the interconnect technologies for this communication was a major challenge faced by OpenCPI.

The appropriate data-plane and data transfer drivers in OpenCPI were optionally loaded when an application's system targeted such hardware and drivers. The driver for a processor connection within a system was then deployed. Which driver was used for a particular inter-component connection was transparent to the component source code.

3.2.1 OFED/Infiniband for Component Data Transport.

In 2009 more than 40% of the Top500 systems [14] used the Optical Fabrics Enterprise Distribution (OFED)/Infiniband for their server and storage connectivity. The OFED is an important embedded application that utilizes a cluster approach with rack mounted servers. This application is connected with either high speed switched Infiniband or Internet Wide Area RDMA Protocol (iWARP) interconnects that execute at speeds of up to 40 Gb/s. The Network Interface Controllers (NICs) for the Infiniband implementation include remote DMA in the hardware that is maximized for performance. Also the application is designed to use a minimum number of processors. This project supported OpenCPI development for such interconnects by allowing its components to efficiently communicate between the processors and/or the subsystems that were connected by this class of hardware.

The OFED enabled drivers and APIs to be selected to effectively target this class of interconnect hardware. This targeted application was an open source distribution of hardware-assisted RDMA hardware drivers and a thin software layer called Infiniband Verbs which unified support across the vendors. The layering was similar to that of OpenCL, in that a thin layer was added to simplify the normalization behavior across the vendors without significantly adding to the overhead and abstractions. OpenCPI's model of message passing between the components was mapable to this layer.

The Linux-based, open source OFED is a broad distribution of the RDMA switch fabric, and the cluster-related software includes layers, drivers and tools. The hardware drivers include those from Mellanox, Hewlett Packard, Cisco, Intel, and Chelsio. The middle layer verbs that were targeted exploited hardware vendor drivers. This same layer was targeted by other middleware for parallel computing, such as the Message Passage Interface (MPI).

The test environment included a software-based low-level driver Soft-RDMA over the Converged Enhanced Ethernet (Soft-RoCEE), which emulated a hardware-assisted Network Interface Card (NIC) that executed Infiniband protocols over link-layer Ethernet hardware. The underlying driver did not adversely impact the OpenCPI support code, and as with other OpenCPI dataplane drivers, had no impact on the components' source codes.

The driver enabled communications between the components executing on a RedHat 64-bit system and a Community Enterprise Operating System[21] (CentOS) 32-bit system. Since the verb header file was common across the vendors, the OpenCPI driver was compiled regardless of the hardware and drivers that were available in the system. Also, all of the versions of the OFED were compliable.

3.2.2 Datagrams for Data Transport over Unreliable Protocols.

The basic communication model for the OpenCPI data-plane drivers was the Remote Direct Memory Access (RDMA), where a process tasks the underlying API to transmit data at some offset from a local memory to a remote memory area that is at some other offset. The DMA interconnect fabrics included PCI Express and RapidIO. The RDMA APIs included the OFED/Infiniband Verbs. The underlying software and hardware layers guaranteed delivery of the RDMA transfers.

Prior to this project, OpenCPI was a simple sockets-based RDMA data-plane driver which transmitted messages over Transmission Control Protocol (TCP)/ Internet Protocol (IP) sockets. Although TCP/IP was said to guarantee delivery, this sockets-based data-plane driver issued a RDMA command descriptor to send the data to the TCP/IP socket, and then assumed that the delivery occurred at the other end.

Ethernet support was developed for communications between an FPGA process and a Linux/x86 process for when it was infeasible to use TCP/IP. However, this protocol was too complex to implement in most FPGAs, and required an unreliable datagram-based RDMA transfer protocol for OpenCPI.

When either a TCP stack or an User Datagram Protocol (UDP)/Internet Protocol (IP) is available, the UDP provides datagram services. Some FPGAs also directly implemented an UDP. Regardless of the underlying datagram service, mapping the RDMA onto an unreliable datagram service was required. This protocol was defined as the DataGram (DG)/RDMA. It enabled datagram frames to carry multiple RDMA-write requests, and provided for frame-level acknowledgements and retransmissions. The RDMA-over-Converged Enhanced Ethernet (RoCEE) datagram protocol defined by the Infiniband Trade Association was too complex to implement in the time frame of the project. This is a good candidate for future efforts since there is hardware support on x86 processors that implements it within the Mellanox Ethernet NICs.

[21] A free operating system distribution based upon the Linux kernel.

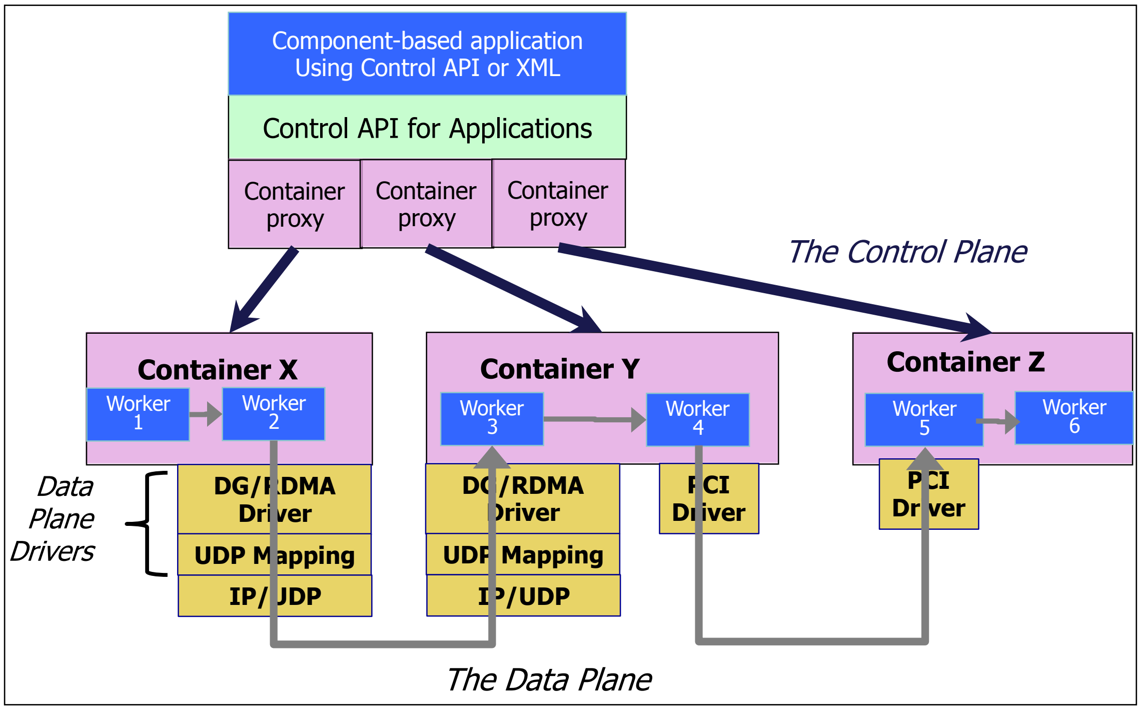


Figure 6. DG/RDMA Driver Layers

To test the software implementation of the DG/RDMA protocol, it was first layered and mapped on top of the UDP/IP. This provided a wide-area-routable datagram capability when an IP stack and routing is available, and is a logical implementation when both sides of the inter-component.

To test the software implementation of the DG/RDMA protocol, it was first layered and mapped on top of the UDP/IP. This provided a wide-area-routable datagram capability when an IP stack and routing is available, and is a logical implementation when both sides of the inter-component conversation are implemented in software. This layering over UDP was accomplished in 200 SLOC and served as an easy-to-debug test environment for the basic software DG/RDMA implementation, which was 1000 SLOC.

This effort not only provided a test-bed for the DG/RDMA protocol definition and software implementation, which was also useful as a mechanism that any inter-component connection could be assigned to. This protocol layering is illustrated in Figure 6.

3.2.3 Ethernet Link Layer DataGram.

To provide an OpenCPI data-plane connection for the Ettus N210 platform that was required by the FSK demonstration platform, a protocol was developed for Ethernet links so that the FPGA-based components executing on the N210 could communicate with the components executing on the host x86 machine. The host executed Red Hat Enterprise Linux (RHEL) 5. Since the use of minimum resources was within the spirit of SWAP, this was implemented on a Xilinx Spartan 3A-DSP-3400 FPGA on the N210 platform. The DG/RDMA protocol was directly implemented over the link-layer Ethernet rather than over the UDP/IP, since this required fewer FPGA resources, would result in a lower latency, and would possibly have higher throughput.

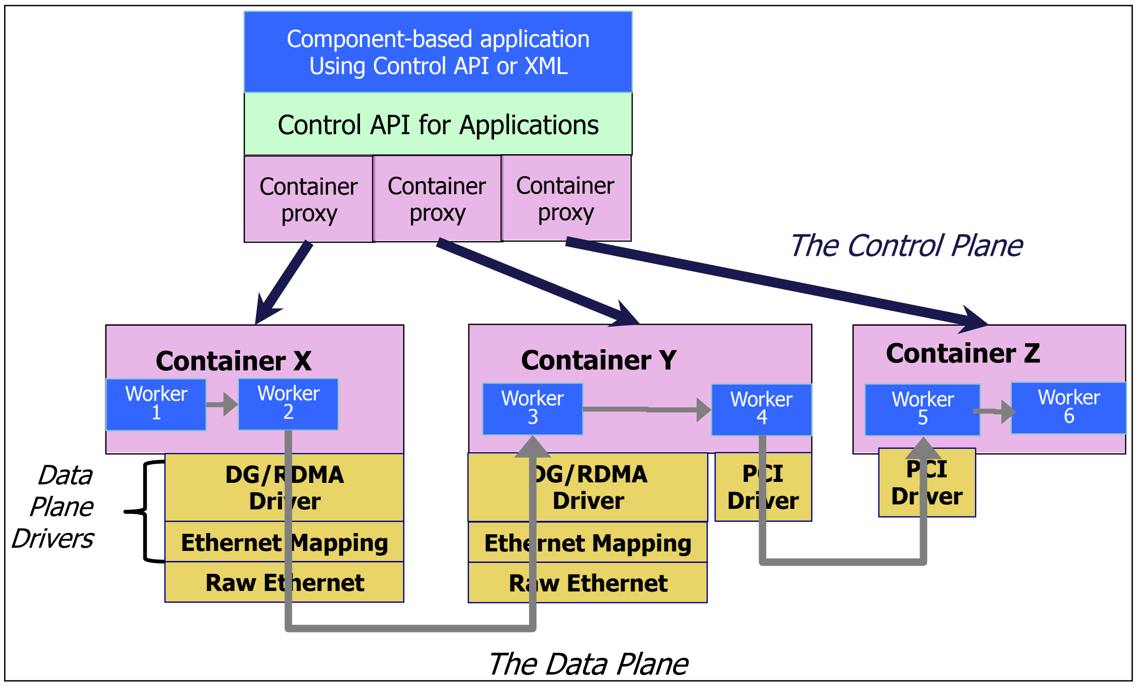


Figure 7. Ethernet-based DataGram DataPlane Driver Layering

This required an alternative software mapping from the DataGram (DG)/RDMA directly onto the link layer Ethernet, and this entirely bypassed the IP stack. This alternative mapping of the DG/RDMA protocol to the Ethernet in software was approximately 200 SLOC, and also required either executing all of the OpenCPI software in the Sudo[22] mode with root privileges, or using the OpenCPI Linux kernel driver. The OpenCPI Linux kernel driver was enhanced to provide a privileged path directly to the Ethernet to avoid the privilege requirement. This completed the software aspects of using the DG/RDMA protocol over the Ethernet link layer, and either UDP/IP or direct Ethernet could be used to transport the DG/RDMA datagrams to support datagram-based RDMA between the OpenCPI components. In Figure 7 is a diagram of an Ethernet link layer, which was an alternative to UDP/IP.

After the software driver and mapping were tested between two computers on an Ethernet link, the DG/RDMA protocol was implemented in FPGA code to execute on the Ettus N210 FPGA. While an UDP/IP implementation was also possible on the FPGA, this was not needed here since the Ettus N210 included an attached front end.

3.2.4 DDS for External Connections of Component Ports.

Although OpenCPI provided an environment where component applications spans a heterogeneous set of processors connected by a heterogeneous set of interconnect structures, there are important cases where components will need to communicate outside this environment rather than to just other components inside that environment. In OpenCPI, an application's component ports are connected to the external services. This is illustrated by comparing one application that has internal connections between its components with another application that publishes data through an external connection into a separate

[22] Sudo is a program for Unix-like computer operating systems that allows users to run programs with the security privileges of another user.

middleware regime. In OpenCPI, external connections are achieved via API or XML by providing a Universal Resource Locator (URL) for the external connection. This indicates that a particular component port in the application should be connected to the communication service indicated by the URL.

The Data Distribution Service (DDS) is an industry standard from the Object Management Group (OMG), the same organization that standardized the CORBA and the Unified Modeling Language (UML). DDS provides efficient local and/or wide area publish/subscribe messaging with control over the quality of service and the data refresh rate. Since it is used in several DoD applications, it was a natural choice for the first supported external connection service for OpenCPI. It is especially useful for dissemination to many users/subscribers when multicast communication patterns are preferred. It also allows the applications using it to be written in multiple languages.

This project implemented both the external connect capability and the DDS external connection capability. DDS support was implemented so that a component's output could be a DDS publisher that disseminates to external DDS-based subscriber applications without changing either the component source code or the compiled code. Also, a component's input can be a DDS subscriber that receives input from an external DDS-based publisher application without changing the component source code or the compiled code. DDS can also be used for transport inside a distributed OpenCPI application, especially for distributed fan-out/multicast configurations. This is another transport way for the components to communicate with each other.

A major challenge in implementing support for external DDS connections was the data interoperability between the OpenCPI messages and DDS. DDS applications publish/subscribe to topics which have CORBA IDL defined data types/structures. The OpenCPI component and connected external DDS applications must have a common data definition. Two tools were created to interconvert the data type metadata between OpenCPI and DDS. Since both OpenCPI and DDS used the basic data type abstractions of IDL as are used in CORBA, this was possible. This allowed the external DDS connection driver to efficiently interconvert the data message formats at runtime. In many cases a conversion was not required. However, if required, the two interconverting tools used are illustrated in Figure 11. Combining these two tools enabled the creation of an OpenCPI component to interoperate with an existing DDS application, and a DDS application to interoperate with an existing OpenCPI component.

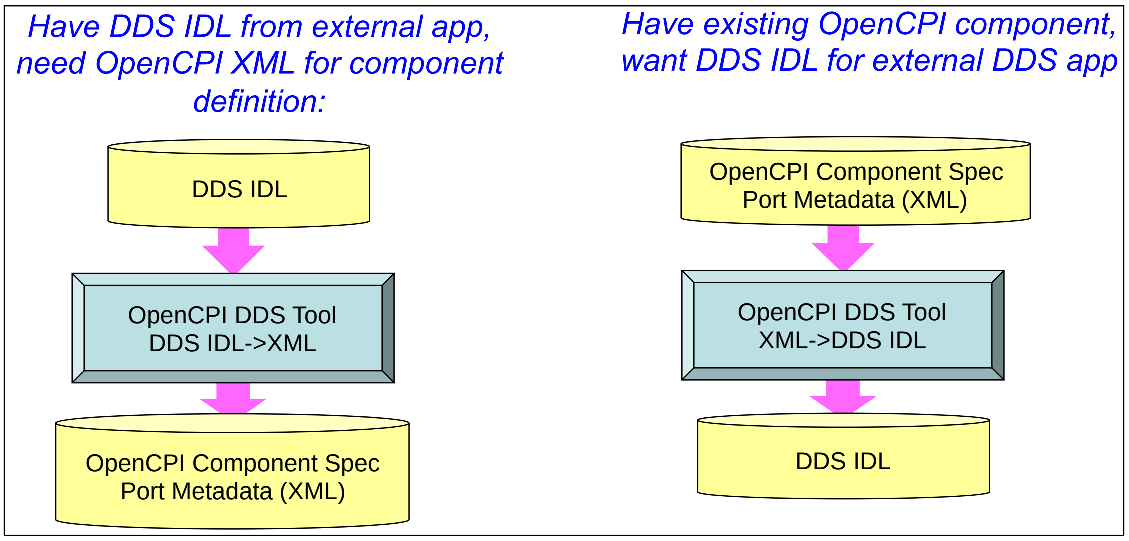

Have DDS IDL from external app, need OpenCPI XML for component definition:
DDS IDL
OpenCPI DDS Tool DDS IDL->XML
OpenCPI Component Spec Port Metadata (XML)
Have existing OpenCPI component, want DDS IDL for external DDS app
OpenCPI Component Spec Port Metadata (XML)
OpenCPI DDS Tool XML->DDS IDL
DDS IDL

Figure 8. Tools for Interconverting Metadata between OpenCPI and DDS

The DDS developments used the open source community licensed version of PrismTech's OpenSplice DDS [15]. This implementation approach for the OpenCPI DDS external connection driver, as shown in Figure 8, avoided requiring the generation of metadata-based code to make connections. The conversion was made in the data path and not with the application level bridging/conversion code, and the amount of the wire protocol code required was reduced from that originally generated by OpenSplice. The URL format for external connections to DDS was developed for this project, since this standard did not previously existed.

3.2.5 CORBA for External Connections of Component Ports.

Implementing a generic CORBA interoperation solution was not an original goal of this effort. During the course of the project it was realized that a key building block for the Software Communications Architecture (SCA) integration experiment was a simple external connection driver for CORBA which used the standard *corbaloc* URLs for CORBA entities in the environment. This also provided a second use case of external connections.

To stay within the scope of the project and the SCA integration experiment, the interoperation for CORBA one-way operations was supported using the OpenCPI RDMA transport drivers. The more general-purpose CORBA Object Request Broker (ORB) transport drivers were not supported.

3.3 Platform Adaptation and Enablement

To use the runtime aspects of OpenCPI on a processing platform, the runtime software modules must support the processors and the processor interconnects. The processor runtime modules were called the containers, and the interconnect runtime modules were called the dataplane drivers. The containers were a combination of software proxies executing with the central core of the application and the actual runtime environment that directly executed the components on the processor. Furthermore, the OpenCPI control plane, which controlled the components, was configured from a central point of the application management. This sometimes required its own type of interconnect support, which was different from the dataplane support. The dataplane support conveyed the messages between the components.

Initially this project had one platform initiative, which was to enable a FPGA processing platform, and was based on the Altera FPGAs in addition to the preexisting support for Xilinx FPGAs. An application trade study introduced a new platform requirement, the Ettus N210.

3.3.1 Altera FPGAs and Altera Stratix4 Development Platforms.

The goals were to enable the OpenCPI HDL infrastructure that had been previously developed for OpenCPI Xilinx FPGAs to be transparently usable for Altera FPGAs. It was necessary to ensure that an application software and the HDL code were not unaware of which type of FPGAs was used, and also that a build flow worked on either of these vendor's tool sets. Also, it was anticipated that other vendors and tool sets would be added in the future. Runtime interoperability between the Xilinx and Altera FPGAs was also developed, implemented and demonstrated.

The two fundamental subtasks were the on-chip infrastructure and tool-build flow. The development of hese two subtasks proceeded in parallel. In Figure 9 is a description of the OpenCPI HDL build flow that was implemented for the Altera FPGAs. Any piece of the HDL code, except for the topmost hardware dependent layers, could be built for both the Xilinx and the Altera FPGAs using the same scripts. Most primitives, workers and application subassemblies could also be built for either family of FPGAs. The full chip files could then be built by targeting a full FPGA platform identifier. Here the Altera Stratix IV development board utilizing PCI Express was selected for experimentation and development.

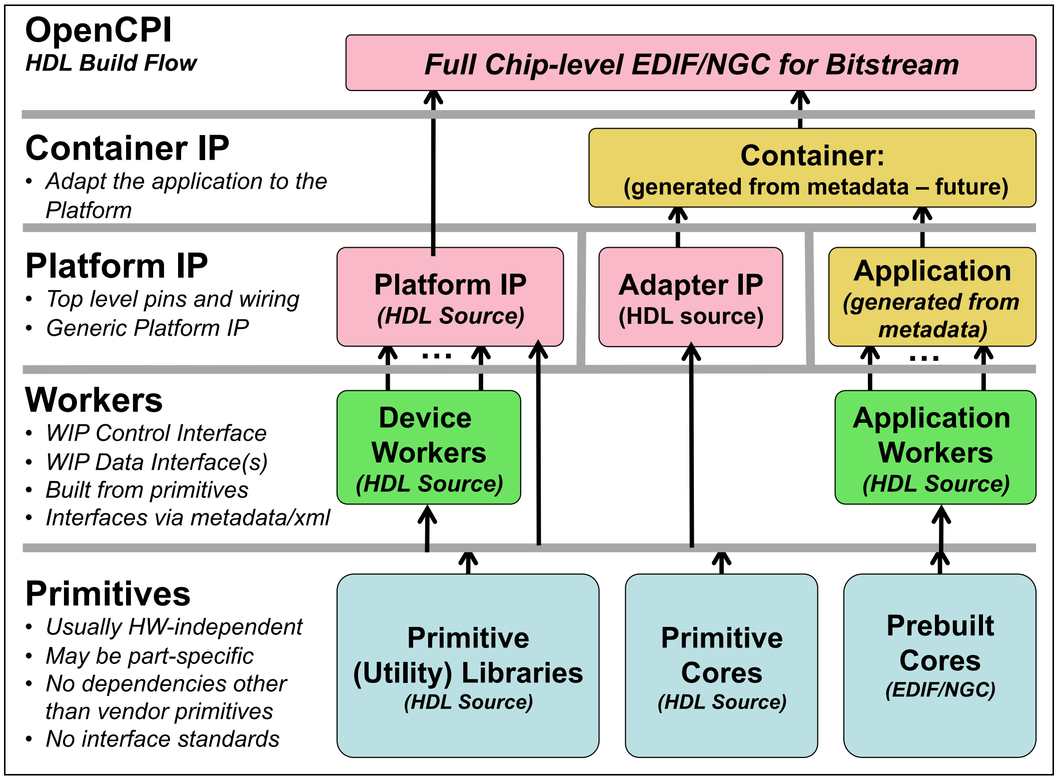


Figure 9. OpenCPI HDL Build Flow

The tooling effort, which was developed in parallel with the hardware-specific efforts, implemented the scripted build flows to create a bitstream chip file. Similar to the existing Xilinx support, the build flow took as its input the specifications to build the chip. The first specification was the target platform, which was a specific FPGA part on a specific board. Also, the application consisted of a connected set of workers, and the container indicated how the application was connected to the infrastructure on the chip.

While this seemed simple, the vendor's scriptable command line tools each operated completely differently. Furthermore, the command line scripting was considered by the vendors to be a secondary usage model in preference to the Graphical User Interface (GUI)-centric modes. The resulting scripts masked the vendor differences.

The platform IP was pre-synthesized with a single undefined reference to the application. The part's application IP was pre-synthesized as a connected collection of workers. Then the two synthesized modules[23] were combined with a container, in a place-and-route step that created the chip level bitstream configuration file. Finally, the metadata was embedded in the bitstream file that notified the runtime library search mechanism of which workers and what connections were implemented in the bitstream.

There were two major challenges for the Altera platform hardware support work in creating the OpenCPI device-dependent infrastructure, which were the attachments to the Peripheral Component Interconnect (PCI) Express fabric and to the Dynamic Random-Access Memory (DRAM). In both cases the primary Altera usage mode was to connect these devices to Altera's own proprietary on-chip infrastructure. There was no example source code for using these devices standalone. Although there was documentation for the standalone usage, that documentation was not accurate.

A third-party hardware board, the HiTech Global PCI Express card with a Stratix 4 FPGA, was also investigated, but the documentation for this was not sufficient, and additional efforts would have been needed to complete this. This proved to be beyond the scope of this effort, and so was not pursued. Additional effort then went into the investigation of a better-supported Altera development board.

The OpenCPI on-chip infrastructure is depicted in Figure 10. Only the Chip Specific Platform Logic required changes for the Altera platform highlighted. The PCI Express, the DRAM in the Device Workers Configured As-required and the Passive Parallel Synchronous (PPS) Time Base were relevant to this effort. Other aspects of the Altera code did not require changes, which confirmed the code's inherent portability.

[23] Netlists contain descriptions of the parts/devices used.

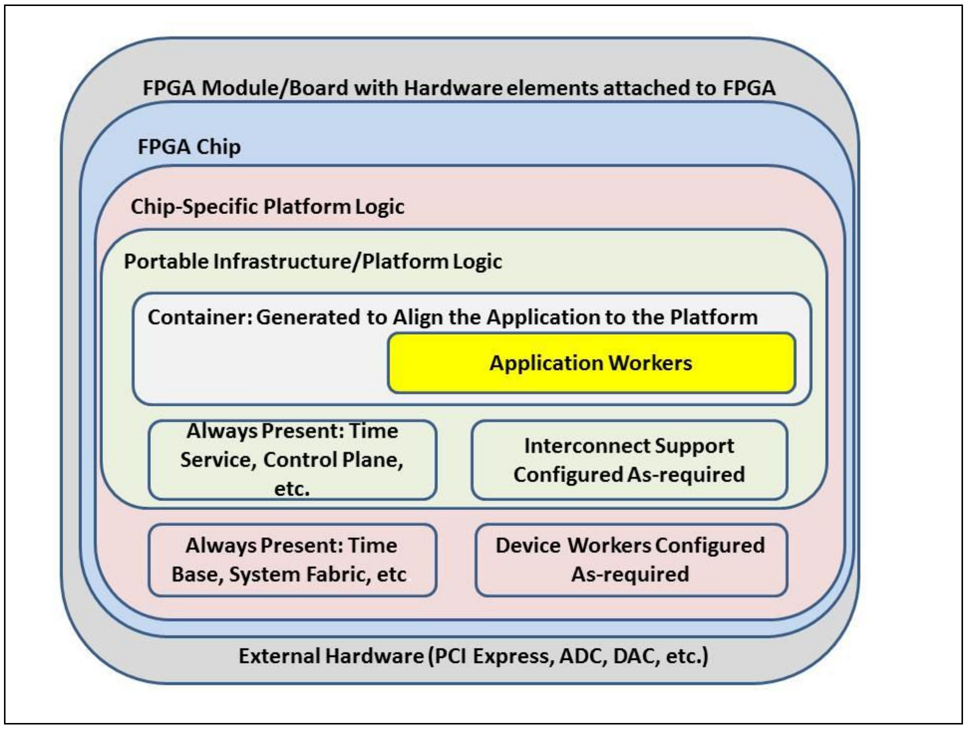

FPGA Module/Board with Hardware elements attached to FPGA
FPGA Chip
Chip-Specific Platform Logic
Portable Infrastructure/Platform Logic
Container: Generated to Align the Application to the Platform
Application Workers
Always Present: Time Service, Control Plane, etc.
Interconnect Support Configured As-required
Always Present: Time Base, System Fabric, etc
Device Workers Configured As-required
External Hardware (PCI Express, ADC, DAC, etc.)

Figure 10. OpenCPI FPGA On-chip Infrastructure

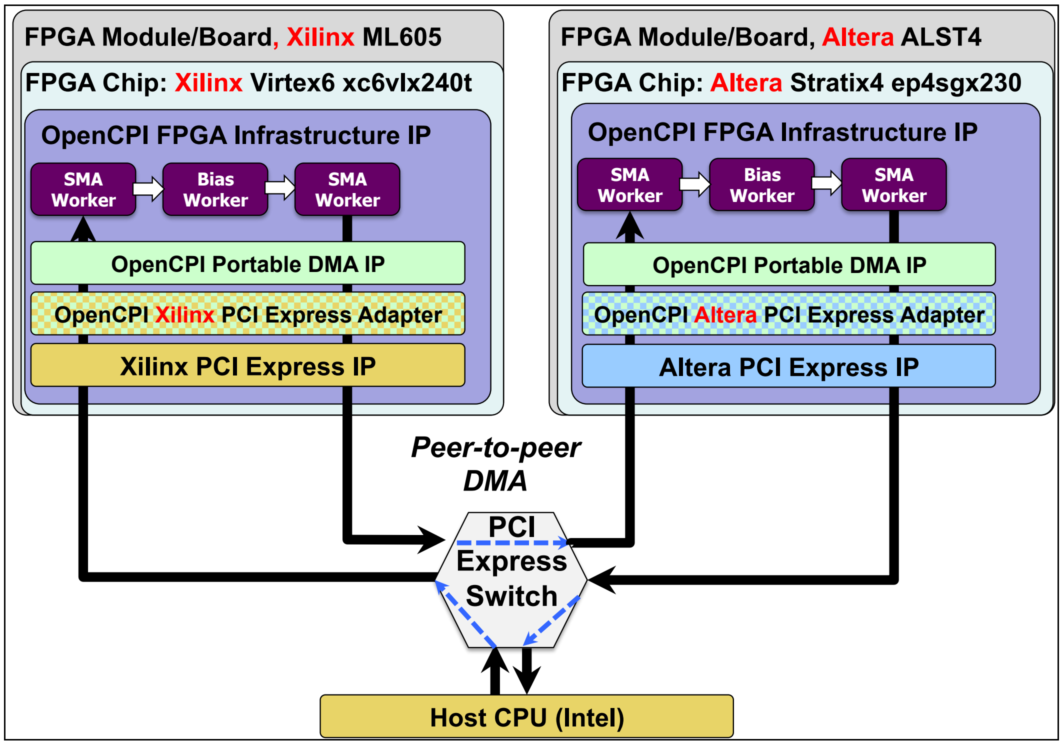

FPGA Module/Board, Xilinx ML605
FPGA Chip: Xilinx Virtex6 xc6vlx240t
OpenCPI FPGA Infrastructure IP
SMA Worker
Bias Worker
SMA Worker
OpenCPI Portable DMA IP
OpenCPI Xilinx PCI Express Adapter
Xilinx PCI Express IP
FPGA Module/Board, Altera ALST4
FPGA Chip: Altera Stratix4 ep4sgx230
OpenCPI FPGA Infrastructure IP
SMA Worker
Bias Worker
SMA Worker
OpenCPI Portable DMA IP
OpenCPI Altera PCI Express Adapter
Altera PCI Express IP
Peer-to-peer DMA
PCI Express Switch
Host CPU (Intel)

Figure 11. Altera/Xilinx Interoperability Demonstration

The final demonstration of this Altera platform and the tool effort was to confirm that in the same system the Altera and the Xilinx boards both interacted directly with each other over PCI Express. This demonstration's configuration is shown in Figure 11. This demonstration showed that the infrastructure was working on both boards, and that the software drives[24] controlled both boards. The DMA and the PCI fabrics were completely interoperable when data was streamed peer-to-peer over PCI Express. Also, the DMA operated both to and from the system memory, and peer-to-peer when bypassing the system memory.

3.3.2 Ettus N210 Platform.

The transition-application study phase was developed and demonstrated on the platform that was the baseline platform for the Ettus Research N210 Universal Software Radio Peripheral application. This application was essentially a radio front end that consisted of a Radio Frequency (RF) tuner/exciter, an Analog-to-Digital Converter (ADC)/Digital-to-Analog Converter (DAC), a FPGA, and Ethernet connections to the system where the software aspects of a radio application were executed on. Since the original application did not utilize a DAC, this part of the platform was later de-emphasized. The Ettus block diagram for this device is shown in Reference [16].

The key platform aspects that supported the OpenCPI development were the FPGA tasks and the software tasks. The FPGA tasks primarily supported the Ethernet interface in the OpenCPI FPGA infrastructure, the Ethernet itself, and the OpenCPI control plane over the Ethernet. Also, the OpenCPI data plane over the Ethernet was supported, as well as the ADC device attached to the Spartan FPGA. Additionally, the basis and the small Spartan 3A Xilinx FPGA were supported.

[24] with no changes for the Altera board

Since this FPGA platform was smaller and more resource-constrained than the previous OpenCPI FPGA platforms, particular attention to the leanness was necessary for all these tasks. This platform was open source hardware, and the technical data and vendor FPGA codes were available for development.

The ADC support leveraged the results of previous ADC efforts on other proprietary platforms. Here though this platform was somewhat different in that the N210 ADC was designed as a quadrature ADC with two ADC channels bonded to provide both In-phase (I) and Quadrature (Q) complex data.

Software support for the control plane and the data plane was added for the raw over link-layered Ethernet. For the existing OpenCPI platforms, this was implemented with PCI Express. Other than the dataplane driver for the datagram-based data-plane support, the major software development involved the discovery and control of the Ethernet-based FPGAs. The associated OpenCPI Linux device driver enhancements enabled raw Ethernet access to the Linux processes controlling the FPGAs via an Ethernet. Device driver development was required to allow the user processors to access the raw Ethernet, which would normally be privileged. Also, it allowed the incoming packets to be processed by the controlled FPGA, which would normally be controlled by higher level protocols such as UDP/IP.

When this work was completed, OpenCPI was able to discover multiple FPGAs attached on a combination of PCI Express and Ethernet in the same system. All platform support was successfully completed, other than the FPGA side of the Ethernet-based data plane, which required some additional effort.

3.4 Integration and Coexistence with Other Frameworks

Real applications sometimes need to combine the aspects of multiple frameworks to obtain all the required functionality. This is especially common when a domain-specific code is available from a framework that does not supply all of the necessary functionality. This project performed two experiments in integrating other frameworks with OpenCPI, both specific to a particular domain. While OpenCPI is certainly not domain-specific, it is focused on embedded systems, with application functionality distributed across heterogeneous processing resources within that system.

3.4.1 Coexistence of OpenCPI with OpenCV.

The Open Source Computer Vision Library (OpenCV) is a library that includes hundreds of computer vision algorithms that are written in C++. This large repertoire of computer vision and image processing algorithms is a valuable resource for building computer vision and other image processing applications.

Originally it seemed that wrapping OpenCV algorithms as OpenCPI components could be combined with other OpenCPI components, and then other technologies could be replace on a component-by-component basis. However, OpenCV algorithms were not created in a component model. Since OpenCV frequently used polymorphic data typing, where code was present to process and to interconvert multiple data types, three integration scenarios were followed. The first scenario was to port code in OpenCV algorithms to form new OpenCPI components. The second scenario was to combine OpenCV algorithms, and in particular image I-Os and format conversions, with the OpenCPI components in the same application. Finally, several applications were created that combined the OpenCV library with the OpenCPI framework.

This resulted in twelve new OpenCPI components [17] with software RCC implementations that were added to the OpenCPI component library, and three new applications that were added to the OpenCPI repository of example applications. This work was done in conjunction with a sponsored graduate student at the MIT Laboratory for Computer Science [7].

The algorithms ported from the OpenCV library into the OpenCPI component library were *Sobel*, *Sobel_32f*, *Canny*, *Blur*, *Erode*, *Dilate*, *Gaussian_Blur*, *Optical_Flow*, *Scharr*, *Good_features_to_track*, *Min_eigen* and *Corner_eigen*. These components were combined into three application examples that directly used the OpenCV library for image conversion and I-O reading from files and displaying in windows, and used the newly created OpenCPI components for image processing. The Canny Edge Detection application (Figure 12), the Feature Detection application (Figure 13) and the Optical Flow application (Figure 14) are shown in the following three diagrams, including which components that were used in each of the applications, and how they were connected.

This process proved to be a good demonstration of the OpenCPI framework, and resulted in several enhancements and bug fixing to OpenCPI. In particular, the optical flow application was used to stress the multiprocessing/multi-core execution of OpenCPI.

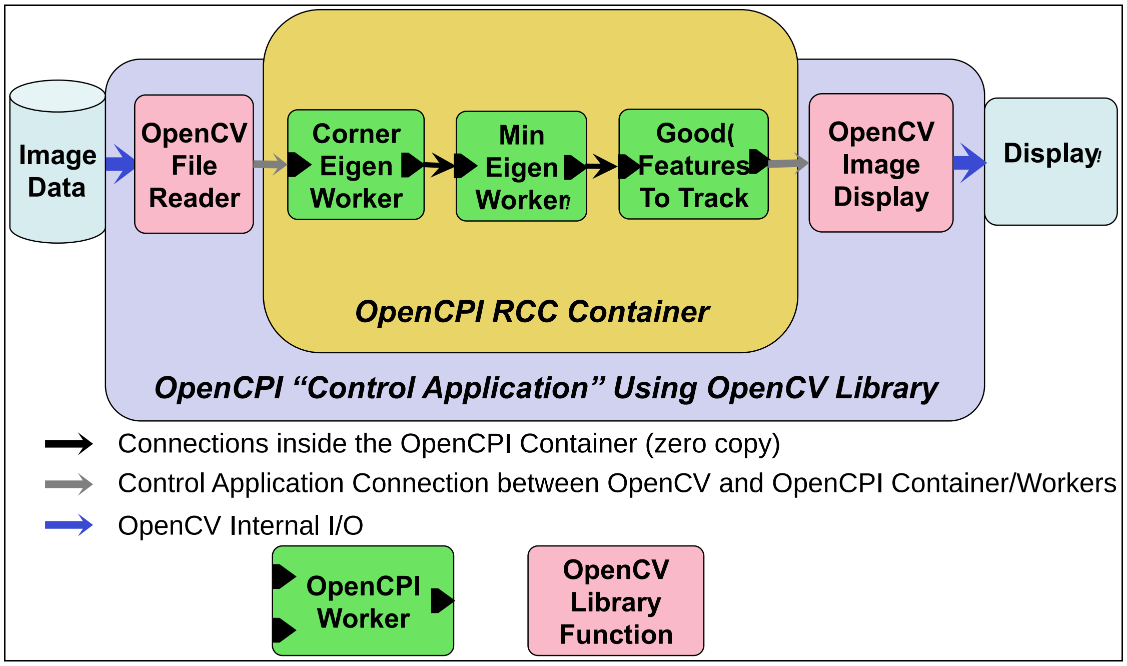


Figure 12. Canny Edge Detection Application

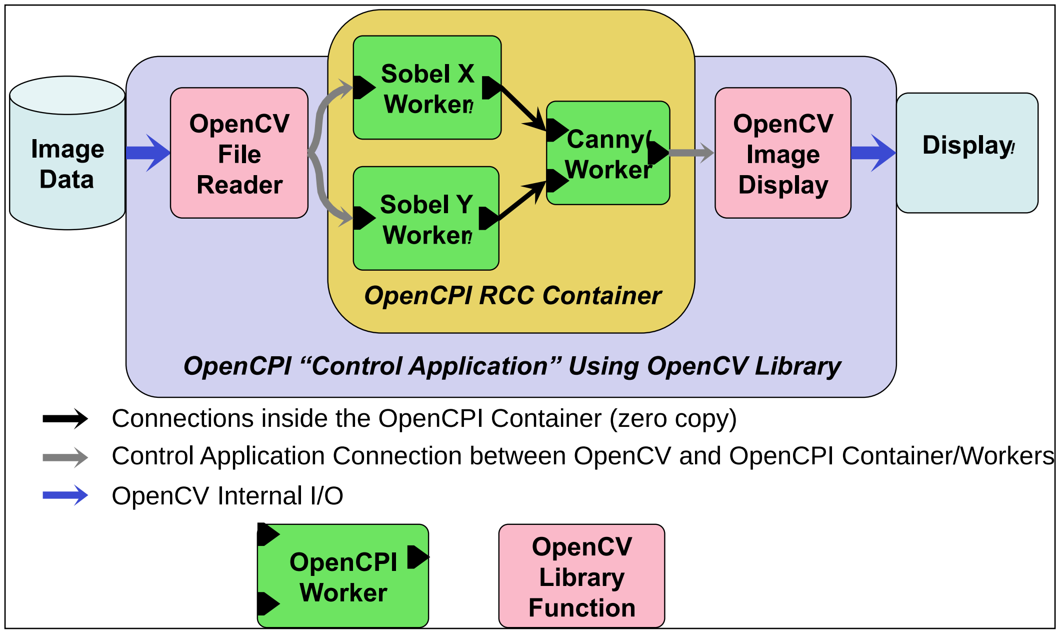


Image Data
OpenCV File Reader
Sobel X Worker
Sobel Y Worker
Canny Worker
OpenCV Image Display
Display
OpenCPI RCC Container
OpenCPI "Control Application" Using OpenCV Library
Connections inside the OpenCPI Container (zero copy)
Control Application Connection between OpenCV and OpenCPI Container/Workers
OpenCV Internal I/O
OpenCPI Worker
OpenCV Library Function

Figure 13. Feature Detection Application

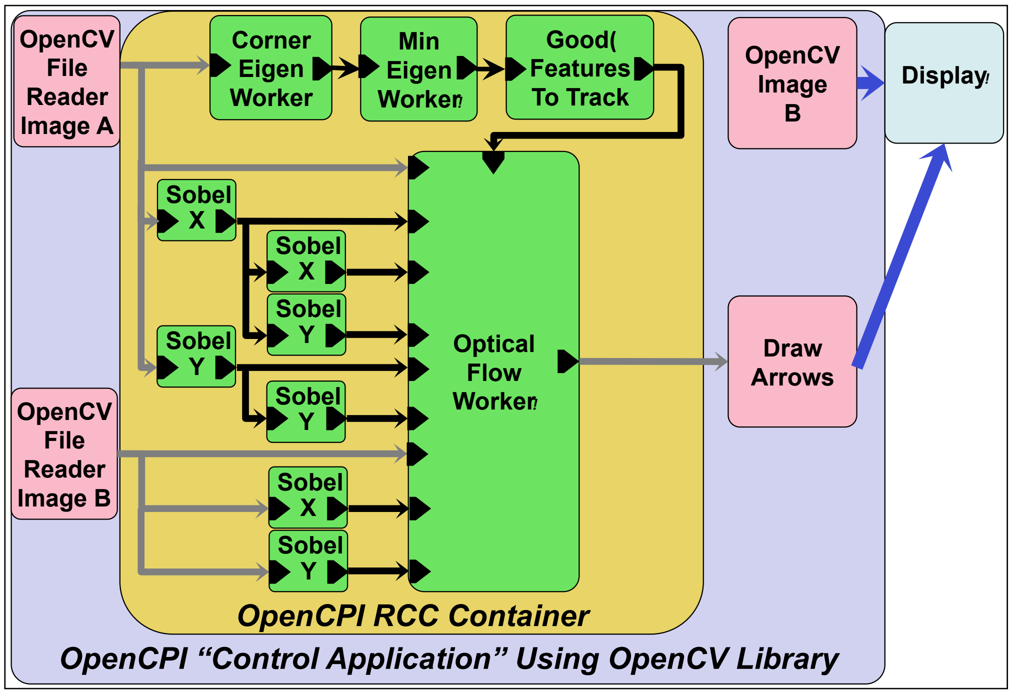


Figure 14. Optical Flow Application

3.4.2 OpenCPI with SCA.

Software Communications Architecture (SCA) is a DoD-developed[25] framework for developing and deploying software defined radio applications. It is a component-oriented framework for software-based components based on using the CORBA for communications, and the Portable Operating System Interface[26] (POSIX) for operating system and runtime library services. It was designed for the software radio domain, which are typically deployed in embedded radio systems.

Prior to becoming open source software, OpenCPI was used inside of and was dependent on an SCA environment. During the open sourcing process this link was eliminated, which resulted in the OpenCPI version being independent of the SCA. Also, the SCA had technical attributes that were rejected as inappropriate[27] for most embedded systems. This project took a new approach to this integration, and investigated whether OpenCPI could be efficiently layered underneath an existing SCA environment without the specific application giving up its inherent efficiency.

A SCA framework was selected that was open source and depended upon the open source CORBA middleware. The selected SCA framework was Open source Software Communication Architecture Implementation: Embedded (OSSIE) framework [18] from Virginia Tech, which was dependent on the open source omniORB CORBA layer[28]. Virginia Tech was a subcontractor that participated with this integration.

In the chosen integration approach OpenCPI did not replace the SCA core framework, which was the management and control center of the SCA environment. However, the key integration elements that were implemented were a SCA Core Framework (CF) *ExecutableDevice*[29] object for each of the implemented OpenCPI containers, and a resource object that was an efficient control proxy for each worker. The relationships of these elements are shown in Figure 15.

The integration of OpenCPI was controlled by and represented to the SCA control system. The OpenCPI containers appeared as SCA executable devices, and the OpenCPI workers appeared as SCA resources, regardless of whether these containers were executing on a FPGA, a GPU or a GPP.

Another important aspect of the SCA/OpenCPI integration was the data plane over which the components communicated with each other. In the SCA the CORBA middleware was used for this. If two OpenCPI workers were communicating with each other under the SCA control framework, these workers communicated directly and efficiently with each other without using CORBA middleware. However, under an SCA framework with OpenCPI, when an OpenCPI worker communicates with a normal SCA component, this additional middleware was necessary.

[25] It is mandated for some programs.

[26] A family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

[27] e.g. CORBA

[28] a robust high performance CORBA ORB for C++ and Python

[29] Framework of operations for executing/terminating processes/threads on a device.

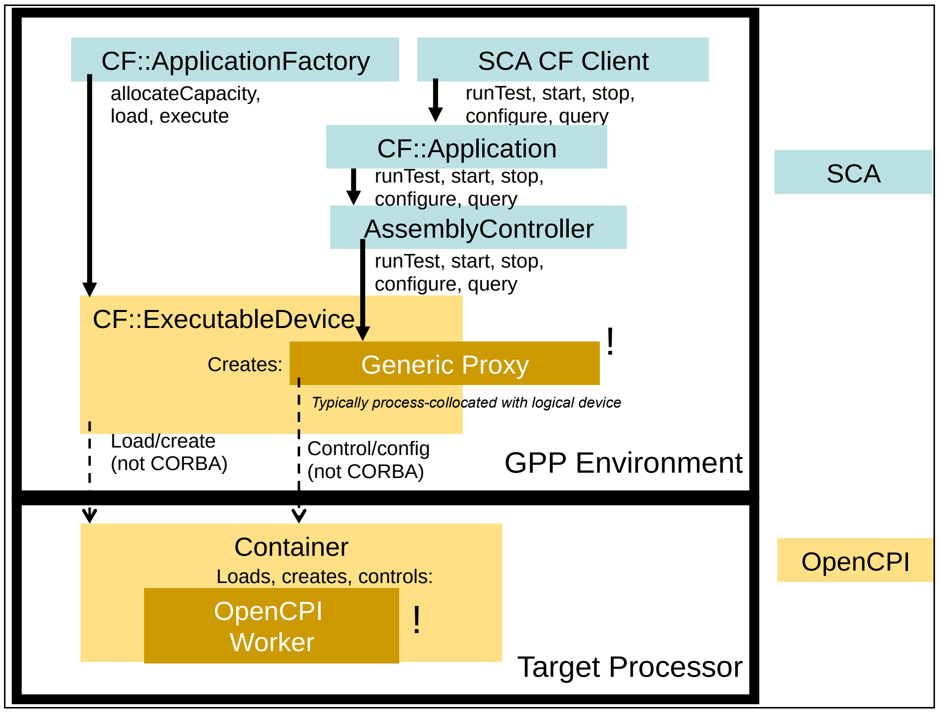

CF::ApplicationFactory
SCA CF Client
allocateCapacity, load, execute
runTest, start, stop, configure, query
CF::Application
runTest, start, stop, configure, query
AssemblyController
runTest, start, stop, configure, query
CF::ExecutableDevice
Creates:
Generic Proxy
!
Typically process-collocated with logical device
Load/create (not CORBA)
Control/config (not CORBA)
GPP Environment
SCA
Container
Loads, creates, controls:
OpenCPI Worker
!
Target Processor
OpenCPI

Figure 15. SCA Layered on OpenCPI

In the approach represented in Figure 16 the efficiencies[30] of the OpenCPI data plane were not affected in the OpenCPI environment and an adaptation from the OpenCPI dataplane to the CORBA middleware layer was only needed when the CORBA-based SCA component was executing. This adaptation was accomplished by writing a special transport plug-in for the omniORB[31] CORBA middleware which, when the SCA component was communicating with an OpenCPI worker, achieved this using the lower level drivers of the OpenCPI dataplane. The CORBA protocol General Inter-ORB Protocol (GIOP) headers were synthesized at the receiving end.

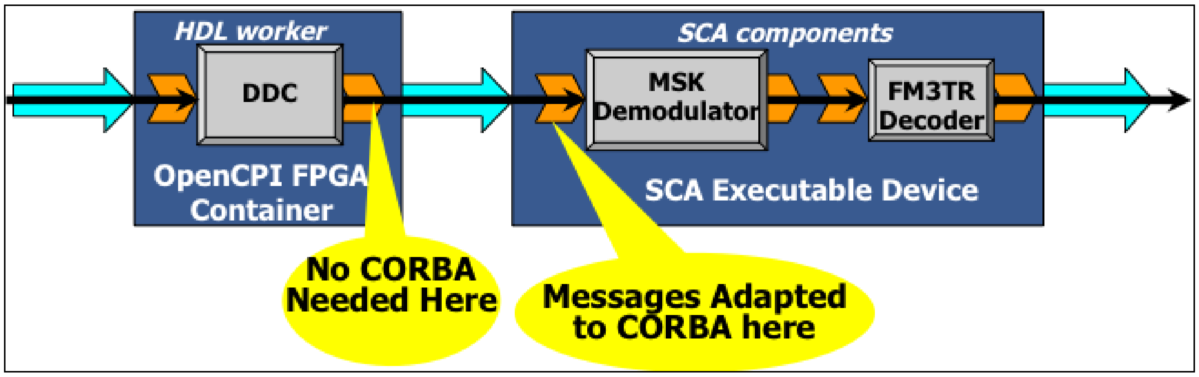


Figure 16. Data Plane Adaptation in SCA Environments

[30] e.g. zero copy and DMA

[31] omniORB is a CORBA object request broker for C++ and Python.

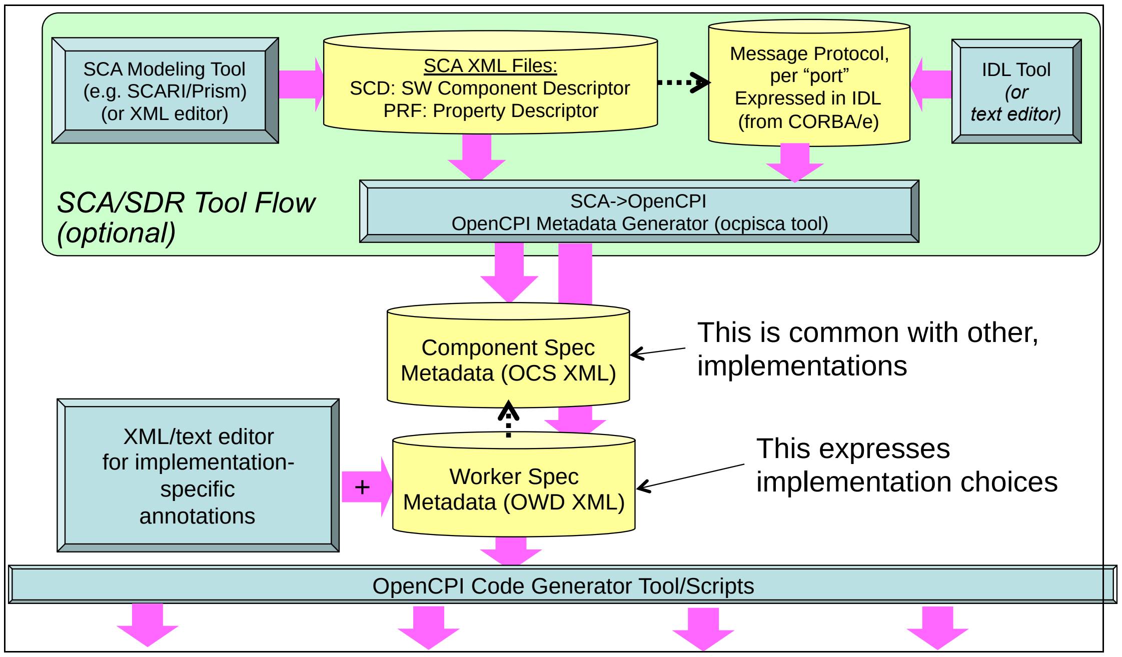


Figure 17. SCA - OpenCPI Metadata Generation

An example of the data plane interoperability solution was where an FPGA worker produces a simple First In/First Out (FIFO) message burst at its data port that was not aware of what it is communicating with, and the receiving SCA component received a CORBA method invocation at its data port. This ORB transport plug-in approach had the additional benefit that the CORBA communications in the system necessitated these drivers even when both sides of the communications were CORBA middleware objects[32].

The tools and the metadata must also be considered when integrating OpenCPI with the SCA. The XML-based metadata described the components, and this metadata was required in both the development and the runtime environments. When implementing the OpenCPI components to serve as embedded work-a-likes to existing CORBA-based SCA components, the *ocpisca* tool [19] is used to create the OpenCPI component metadata XML files *OCS*, and the *OWD* files from the analogous SCA files *SPD*, *SCD*, and *IDL*. This tool flow is shown in Figure 17.

The opposite direction can be similarly and more easily automated using existing OpenCPI components in an SCA environment by generating the SCA metadata from the OpenCPI metadata. Due to time constraints, this latter automation was not completed in this project, but is straightforward since the OpenCPI metadata is essentially a proper subset of the SCA metadata.

The limitations of this SCA integration approach were that the data port Interface Description Language (IDL) was restricted to the embedded subset as defined by the Object Management Group/International Organization for Standardizations (OMG/ISOs) CORBA/E micro profile. The control ports were SCA standard, but a full control and data proxy was required for further customizations. When two components were hard-wired to each other, OpenCPI pre-connected implementations required a patch to the core framework. Previously the Harris and Software Communications Architecture Reference Implementation

[32] e.g., when there are multiple GPPs on a fabric like PCI Express.

(SCARI) Core Framework (CF), OIS Company, The ACE Orb (TAO), and VxWork’s Real Time Operating System (RTOS) had been addressed. In this effort the open source Software Communication Architecture Implementation Embedded (OSSIE) CF, omniORB, and Linux were addressed.

3.5 Ease of Use Improvements

A number of efforts were made here to simplify the process of developing, configuring and executing applications using OpenCPI. Most of this is related to configuring an application to execute in a heterogeneous processor environment.

3.5.1 Component Libraries in the Search Path.

To execute a logical OpenCPI application component it must be determined on which processor among the available processors the application will be executed on. Also, it must be determined as to which implementation, among available alternatives, should be used. When this project started, these decisions were embodied in a top level main program called the Control Application, in which the APIs specified for each component which binary file was to be executed, and which processor container it would execute on. Thus the application code explicitly and directly specified these two decisions. This was simple at the API level, but when an application was executed in different configurations that used different component implementations, the Control Application was required to implement all the configuration alternatives in an *ad-hoc* code.

The first effort to simplify this process was to have an implicit implementation selection using the very common technique of a library search path. Search paths for executables and libraries are common in many systems including Microsoft Windows, Linux and MacOS. The two environment variables that are typically used are the path variable that indicates the locations of the executables and scripts, and the library variables that indicate the possible locations of the dynamically loaded libraries.

Here these same techniques were developed for OpenCPI. The *OCPI_LIBRARY_PATH* environment variable indicated an ordered sequence of directories that searched for the components’ implementation binaries. This allowed the simplification of the component names, so that these directories were searched for binary files that contained component implementations. The files found were then dynamically loaded on the appropriate processor.

The knowledge of what components were implemented in each file, and what type of processor each would execute on, was embedded in the binary files during the OpenCPI build process. These component library directories were heterogeneous libraries that contained binary files for the different types of processors that included FPGAs, GPUs, different GPPs, and different Operating Systems (OSs). As with any library search path, a library can shadow other libraries by being earlier in the path. For example, a library full of FPGA implementations might represent some accelerated versions of a more general-purpose software-only library containing default software implementations of some of the components.

While component binaries represented one component implementation for one type of processor, the component binaries actually contained multiple implementations for the same or different components. These binaries were each written for a specific processor and that processor’s OS. Finally, a library directory in the search path was recursively searched for the component binaries, so that an *ad hoc* directory structure containing files and directories was a library.

The search process was optimized by caching the metadata in all of the component binary files that were found in the path, so that a search for each component in the application could be rapidly completed. The directories were only scanned once at the program startup. Further optimizations were possible to save cache in the file system, which avoided directory searches during each program’s execution.

In a Linux shell, the following line of code set the component library search path:

```
   export OCPI_LIBRARY_PATH=/home/me/fpga-impls:/syst/software-impls .
```

3.5.2 XML Applications without C^{++} Coding.

The existing OpenCPI application control interface was a C^{++} API that specifically created and managed the containers, applications and workers. Additionally, it connected and configured the workers. In the main program the API programmatically constructed the application in the C^{++} control application. This resulted in a control-application/main-program of several hundred lines of code. A newer, simpler, more compacted and less error-prone XML-based description of applications was developed, and was a significant ease-of-use enhancement.

The XML application format consisted of a top-level application element, with instance and connection child elements. A copy application consisting of file reading and file writing workers is:

```
<application>
   <instance worker='file_read'>
      <property name='filename' value='hello.file'/>
      <property name='messageSize' value='4'/>
   </instance>
   <instance worker='file_write'>
      <property name='filename' value='out.file'/>
   </instance>
   <connection>
      <port instance='file_read' name='out'/>
      <port instance='file_write' name='in'/>
   </connection>
</application>    .
```

This example had two workers, the *file_read* and the *file_write*, with the out-port of the *file_read* connected to the in-port of the *file_write*. The filename property of the *file_read* worker was initialized to the *hello.file*, and the filename property of the *file_write* worker was initialized to the *out.file*.

The more advanced features included a transport attribute of the connection element that indicated which dataplane transport mechanism that was needed for the connection (OFED, TCP-IP, datagram-UDP, etc.). The done attribute of an application element indicated which of the workers would acknowledge when the application was completed. The external child element under the connection elements indicated if there was an external, and not a worker port, aspect to the connection. A value file attribute of the property element contained the name of the file that contained the textual initial value for the property. The value attribute contained the textual value itself.

This XML application was used as a part of the new API and the new utility program. This new API was added to allow a C^{++} control application to create an Application object from an XML string. This was a C^{++} constructor for a new Application class. After construction, this new class included the initialize, start and wait operations. An example of a main program using this new XML-based C^{++} API was:

```
int main(int argc, char **argv) {
   std::string
      hello("<application>"
            "  <instance worker='file_read'>"
            "    <property name='filename' value='hello.file'/>"
         "  </instance>"
            "  <instance worker='file_write'>"
            "    <property name='filename' value='out.file'/>"
            "  </instance>"
            "  <connection>"
            "    <port instance='file_read' name='out'/>"
            "    <port instance='file_write' name='in'/>"
            "  </connection>"
            "</application>");
   OCPI::API::Application app(hello);
   app.initialize();  // obtain all resources
   app.start();       // start execution
   app.wait();        // wait for workers to be finished
   return 0;
}
```

This C^{++} API allowed the program to construct and manipulate the XML prior to execution.

The other way that this new XML application was used was by putting the XML into a file, and using a new utility program called *ocpirun* to execute it. When the above XML was in the file *copy.xml*, this shell command executed:

```
ocpirun copy.xml .
```

This utility program included the two options listed in Table 2.

Table 2. Utility Program Options

Option	Description
-d	Dump all properties of all workers after execution
-v	Be verbose, describing the progress of execution

The combination of the XML form of application description, the C^{++} API for launching and controlling such applications, and the utility program for executing them provided a simpler way for applications to be constructed and executed. It is expected that this method will be the preferred one for future OpenCPI applications.

3.5.3 Scoring Preferred Implementations.

A specific capability for choosing the best component implementation was also developed. Building on the previous enhancement of searched and cached component libraries, the application specified an optional implementation scoring selection expression for each of the application's components. The purpose of this expression was to specify the optimal choice among the alternative implementations when more than one implementation was available. This capability also provided a way to specify the minimum conditions for an acceptable implementation choice.

The expression was a normal expression in the syntax of the C language, with all the normal operators. Logical expression *a == 1* returned 1 on true, and 0 when false. The variables in the expression were either component property values that had fixed implementation values, or special built-in identifiers that indicated well-known implementation attributes. The special identifiers are listed in Table 3.

Table 3. Special Identifiers

Identifier	Purpose	Example
Model	Implementation Authoring Model name	rcc
Platform	Platform the implementation is built for	x86_64 or ml605
OS	Operating system that the implementation is built for	Linux

The value of the expression was considered an unsigned number, where a higher number was preferred to a lower number, and zero was considered unacceptable. If the expression for an implementation had a zero value, that implementation was not considered acceptable. A simple example is:

```
model==’’rcc’’ .
```

This indicates that the model was *rcc*, since otherwise the expression's value would have been zero. The example of

```
error_rate > 5 ? 2 : 1
```

indicated that the error rate property is relevant. If the error rate property is greater than 5, it is better than if it were less than or equal to 5. However, a positive integer of equal to or less than 5 is still acceptable.

If there was no selection expression, then the score of the implementation would be 1.

Using the OpenCPI application control API, the implementation scoring expression is provided as the selection parameter in the *createWorker* API. In the XML application specification that is described in detail below, the selection expression is provided as the optional selection attribute of each component instance. This application code

```
<application>
  <instance worker=‘‘psd" selection="latency < 5"/>
  <instance worker=‘‘demod" selection="model=='rcc'"/>
```

indicated that the *psd* worker required an implementation with latency > 5, and the *demod* instance required an implementation with an Authoring Model of *rcc*. The implementation selection and scoring features simplified the application's preferences and requirements for each of its component.

3.5.4 Property Values for Complex Data Types.

The initial property values in the XML application forms were in a textual representation. While this was trivial for the simple scalar values and strings, it was not available in OpenCPI for the advanced data types such as multidimensional arrays, arrays of structures, etc. The initial OpenCPI capabilities for the property data types were limited to those available in the SCA standard. While this ensured some cross-framework compatibility, it also limited the number of the important use-cases. The enhancement developed here provided an expansion of the possible property data types and the textual representations in both XMLs and in APIs.

The data type enhancements expanded the property data types to the full data structure capabilities of the Interactive Data Language (IDL) originally developed for CORBA and used for DDS. This data typing system expanded beyond the simpler SCA standard. Examples of new added data types for component properties included fixed size arrays, multidimensional arrays, arrays & sequences of structures, recursive arrays, sequences and structures. These were set either directly in the XML value attribute of a property element, placed in a file using the *valuefile* attribute of a property element, or were a C^{++} application utilizing the *setProperty* method of the Application class.

3.5.5 Automating Worker Deployment.

A new capability was developed, implemented and demonstrated that automatically changed the number of processors for an application by applying specific processor allocation policies. The original default policy implemented a first fit algorithm that determined which container a worker would be executed on. This algorithm was guided and constrained by the selection and scoring mechanism that was described earlier. To achieve the ability to automatically change the number of processors that were to be implemented, three policies were followed. Each worker was assigned to a container/processor/core in a round robin fashion. A minimum number of containers were used, which increased the collocation. A fixed maximum number of processors were employed in a round robin fashion.

The new default policy was to first use all of the available containers, and to spread the workers across the containers in a round-robin allocation. Software-based containers [10] originally used one CPU core, but multi-core processors allow the possibility of one container on each processor.

To support a bounded number of processors for the application the *maxProcessors* policy was implemented, which indicated by setting the *maxProcessors* attribute in the application XML to the number desired. For example, to set the maximum number of processors to 3, this code was developed:

```
<application maxProcessors="3"> .
```

Similarly, the *minProcessors* policy reused containers when possible, using the minimum number to execute the application using minimal resources.

3.6 Testing Capabilities

A number of efforts were undertaken to mature OpenCPI by improving the core infrastructure test capabilities. This was extended to include the application and component developments.

3.6.1 Unit Test Framework for Infrastructure Code.

Although there were a number of ad-hoc tests for aspects of the OpenCPI infrastructure, a standard harness was necessary for creating and executing the unit tests of the infrastructure code. A number of open-course unit test frameworks were tested for C^{++}, and the most appropriate candidate was selected.

That test framework was incorporated as a unit test framework, and a number of existing tests were migrated and ported into it.

The motivation for selecting a unit test framework was to improve and enforce the consistency of test development, how the defects were reported, and how the tests were executed. Also, it was desired that there was a reduction in the complexity required to write the tests, which would then encourage more tests to be developed. Candidates needed to be open source software with licenses compatible with OpenCPI, and needed to support C^{++} software development language with exception handling. Also, the candidates needed to be highly portable with minimum environment, OS and other package dependencies. Finally, these open source candidates needed to be actively maintained.

Based on these criteria, the frameworks *Boost.Test*, *CppUnit*, *CppUnitLite*, *Google Test (gtest)*, *Unit*$^{++}$, *NanoCppUnit*, and *CxxTest* were evaluated, and the results are summarized in Table 4.

Table 4. Unit Test Framework Evaluations

	Frameworks						
	Boost.Test	*CppUnit*	*CppUnitLite*	*GoogleTest*	*NanoCppUnit*	*Unit*$^{++}$	*CxxTest*
License	Okay	Good	Good	Okay	Good	Good	Good
New tests	Good	Bad	Good	Good	Good	Bad	Good
Fixture	Okay	Okay	Bad	Good	Good	Okay	Good
Suite	Okay	Okay	Bad	Okay	Okay	Okay	Okay
Exception	Good	Okay	Bad	Good	Bad	Okay	Good
Assert	Good	Okay	Okay	Good	Okay	Okay	Okay
Reporting	Bad	Good	Okay	Good	Bad	Bad	Okay
Portability	Bad	Okay	Good	Good	Bad	Okay	Okay
Activity	Okay	Okay	Bad	Good	Bad	Bad	Bad

From the results presented in Table 4, the *GoogleTest (gtest)* was determined to be the best alternative, as it was actively maintained and the test creation was easy and compact. Also, it had good error and exception reporting. The *gtest* measured the time to execute a test case and reported it, which allowed the determination of execution timings. Continuous Integration Server for Frequent Builds.

To facilitate a community of developers and users, a continuous integration server was selected and configured to build and test OpenCPI (Figure 18). This server executed the building and testing on several system configurations, and made results available on a web-based dashboard that included the status of OpenCPI. Having such a status dashboard abetted the determination of which versions of code we
uilt and
tes

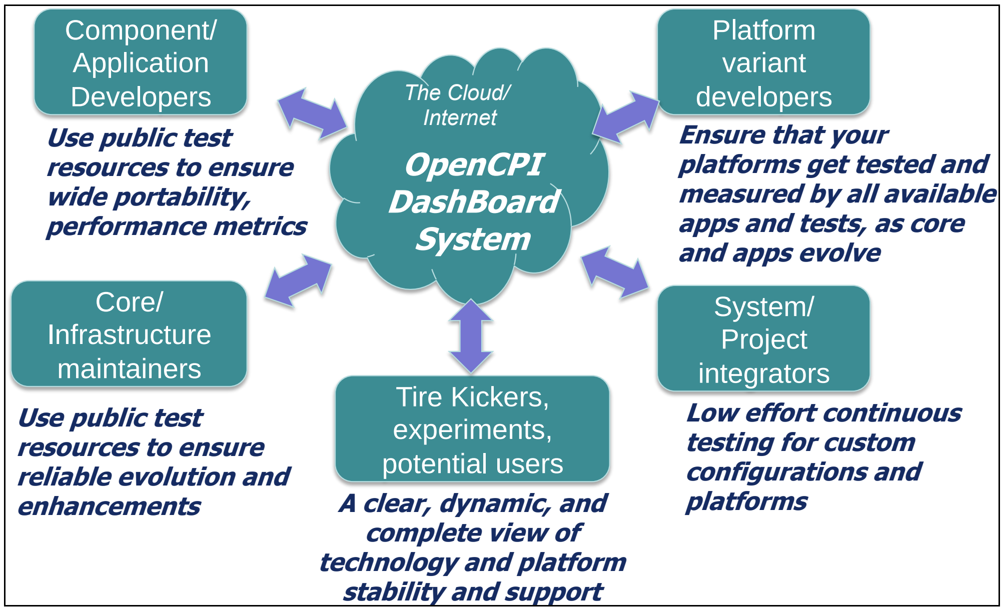
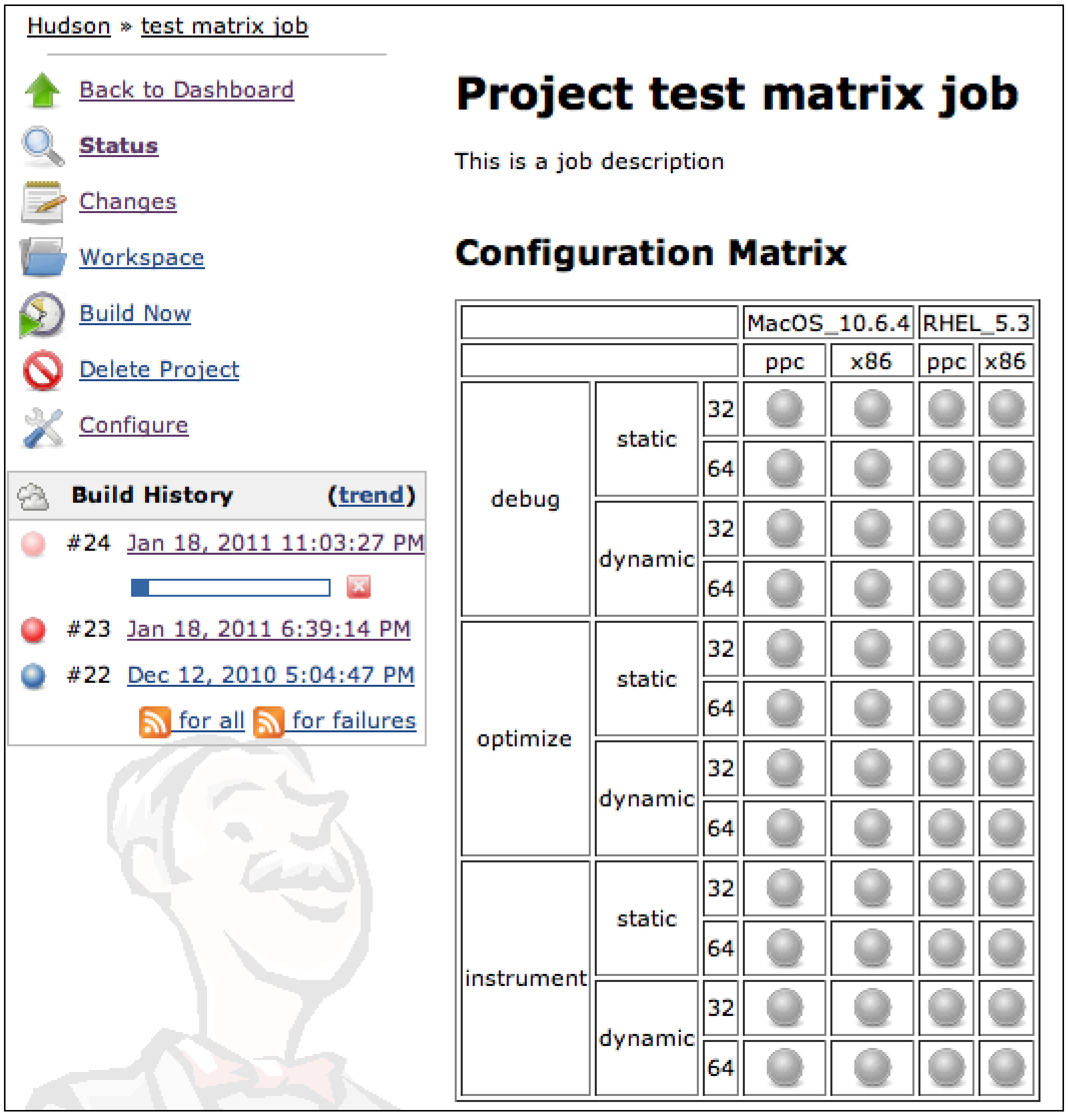

Component/ Application Developers
Use public test resources to ensure wide portability, performance metrics
The Cloud/ Internet
OpenCPI DashBoard System
Platform variant developers
Ensure that your platforms get tested and measured by all available apps and tests, as core and apps evolve
Core/ Infrastructure maintainers
Use public test resources to ensure reliable evolution and enhancements
Tire Kickers, experiments, potential users
A clear, dynamic, and complete view of technology and platform stability and support
System/ Project integrators
Low effort continuous testing for custom configurations and platforms

Figure 18. Continuous Build/Test Dashboard Users

There were no unexpected browser requirements for the viewing and the configuration of this dashboard. The building and the testing can be on different systems. Community members can contribute hardware for performing the builds and the tests. Very heterogeneous system testing was enabled, which included mixing different technologies, different configurations, multiple platforms, and proprietary-custom technologies with multiple platform configurations. Software/component/applications were sometimes tested on systems that were not owned by the tester, and to which the tester did not necessarily have access to.

The Hudson system is an open source continuous integration tool written in Java. Since renamed the Jenkins system [8], the Hudson system met the required criteria, was portable in that it was written in Java, and included within it the concept of multidimensional configuration matrix builds and distributed build slaves. This meant that it automated the traversal of a configuration matrix to build many systems and configurations, and determined the build slave system, which could be located anywhere on the internet. It could also build for any given configuration.

In Figure 19 is presented a screenshot of the web page for viewing a configuration matrix job, in which the individual circles turn red or green as the builds and tests are successfully or unsuccessfully accomplished. Several slave systems are shown from which combinations of systems can be built.



Hudson » test matrix job

Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure

Build History (trend)

●	#24	Jan 18, 2011 11:03:27 PM
●	#23	Jan 18, 2011 6:39:14 PM
●	#22	Dec 12, 2010 5:04:47 PM

for all for failures

Project test matrix job

This is a job description

Configuration Matrix

			MacOS_10.6.4		RHEL_5.3	
			ppc	x86	ppc	x86
debug	static	32	●	●	●	●
		64	●	●	●	●
	dynamic	32	●	●	●	●
		64	●	●	●	●
optimize	static	32	●	●	●	●
		64	●	●	●	●
	dynamic	32	●	●	●	●
		64	●	●	●	●
instrument	static	32	●	●	●	●
		64	●	●	●	●
	dynamic	32	●	●	●	●
		64	●	●	●	●

Figure 19. Continuous Build/Test Dashboard Configuration

One key feature of Jenkins that facilitated a community was that the users could make available a particular system configuration as a building and testing slaves at specific times, i.e. between 2 AM and 4 AM. This resulted in their configurations being available for testing by others when they were not using them.

3.6.2 Unit Test Capabilities for Heterogeneous Components.

As part of the trade study to port the components of the Frequency Shift Keying (FSK) application, it was determined that this process would be accelerated if a common unit test capability was available for components. This resulted in a component tool and method components being developed and included in the unit testing component library.

The *unitTest* utility program created and executed the OpenCPI applications that served as each component's test bench mark. It supported components that were typical signal processing components that had single-input-single-output topologies. Test data was sent to the components, and the output to a known good output data set was compared. The new output was compared with the variance-from-known-good-output.

The options provided by this heterogeneous unit test utility included which Authoring Model was to be tested[33], what the numerical acceptance threshold was, and what other property values to apply to the worker that was being tested. This method tested different implementations of the same component.

The OpenCPI component libraries contained the specification file labeled with an extension *.OWD* in a common subdirectory directory, and then each component implementation would be placed into its own subdirectory named <*component-name*>.<*authoring-model*>. For example, the RCC[34] model of the framegate component was placed in a subdirectory named *framegate.rcc*, and the OpenCL (OCL)-based GPU model was in a subdirectory named *framegate.ocl*. Since the unit test for each component was heterogeneous, it was located in a subdirectory named *framegate.test*. For most components, the only contents in the "<*component*>*.test* directory would be the scripts to execute the *unitTest* utility with the appropriate parameters. When the *unitTest* utility was not used, i.e. when the component had features that required a more customized test bench, this directory would contain the customized unit test program, which was a variant of *unitTest*. Customized test benches then tested the different implementations.

3.6.3 Protocol Monitors for FPGA Components Witten with the HDL Model.

While the unit test tools above served to test the components in a technology-independent manner, and tested whether the resulting data was correct, the HDL implementations executing on FPGAs and in simulators also required testing at a lower level. The external interfaces of the OpenCPI HDL workers utilized the OCP-IP interfacing definitions as profiled for OpenCPI. These interfaces were called Worker Interface Profiles (WIPs), and were defined in the reference manual for the OpenCPI HDL Authoring Model [19]. To confirm that an HDL worker correctly obeyed the wire-level protocols defined in the WIPs[35], the concept and implementation of WIP Protocol Monitors were introduced.

A WIP Protocol Monitor was an FPGA module that was inserted as a passive observer on an interface between two workers, or between the OpenCPI HDL infrastructure and a worker. It verified that the correct signaling and/or monitor events were occurring at the interface, and then the outputted the series of events that it had observed.

Two WIP protocol monitors were implemented, one for the Worker Control Interface (WCI) and one for the Worker Streaming Interface (WSI). A Capture Worker was implemented to capture the event outputs of the protocol monitors. The Protocol Monitors and the Capture Workers are shown in Figure 20.

The Protocol Monitors observed the interfaces either in real hardware or in a simulation environment, and reported the events as output messages on their respective WSI output interfaces. Each monitor was connected to a Capture Worker that time-stamped and stored the events in its respective private memory.

[33] RCC vs. HDL vs. OCL

[34] written in C Language

[35] i.e., the protocol of the data and control signals at each interface.

After execution, the software read the stored events. This information was used to assess any protocol violations and to measure the performance.

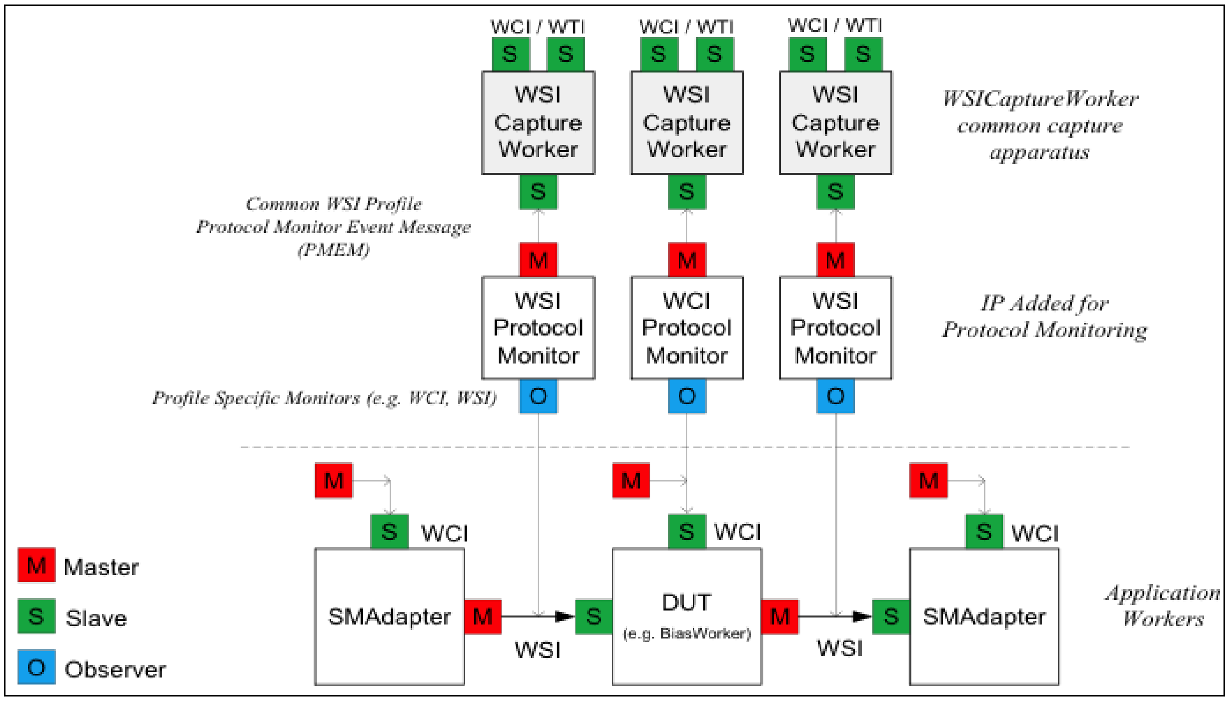


Figure 20. Protocol Monitors and Capture Workers

4.0 RESULTS

This project focused on closing several technical gaps, including issues associated with the ease of use, maturity improvements and application experiments. Each effort tested both the premise of OpenCPI's value and the suitability of the actual existing implementation of tools and runtime. The results were fully captured in the evolved and enhanced version of OpenCPI residing in the Open Source Repository at the *Opencpi.org* website. The OpenCPI tools, runtime infrastructure, components, applications, and documentation were all expanded and improved.

4.1 Performance and Time Measurement

The project requirements motivated the use of tools, many of which were oriented towards understanding the relative timing of events in the parts of the OpenCPI system. Some of these tools were focused on the component and the application developments, while others were focused on the measurement and fine-tuning of the infrastructure. This section describes in more detail these measurement-related efforts.

4.1.1 OCL Authoring Model Measurements.

In the course of developing the OCL-GPU Authoring Model, several measurements of code complexities were made, which were determined in terms of the Source Lines of Code (SLOC). The SLOC is a software metric that measures the size of a computer program by counting the number of lines in the text of the program's source code. The SLOC is one estimate of the programming productivity of a software code. Three application codes were generated with both OpenCL and OCL. The results of the respective SLOC measurements are presented in Table 5.

The three applications that were compared were Vector Addition, 1-Dimensional Fast Fourier Transform (FFT), and Matrix Multiply. The host SLOC for the OCL was technology independent, as the same host code was applied to components written in either model. The host SLOC for OpenCL was specific to OpenCL. Regardless of the components size, the additional lines of the kernel codes and of the XML codes were modest. Recent enhancements have reduced this further.

Table 5. OCL SLOC

	Applications					
	Vector Addition		**1-D FFT**		**Matrix Multiply**	
Code	OpenCL	OCL	OpenCL	OCL	OpenCL	OCL
Host*	277/49	204/19	301/37	200/15	498/51	239/19
Kernel	9	22	905	918	91	104
XML	0	11	0	15	0	11

*The Host is the total SLOC/relevant SLOC.

Table 6. OCL Model Execution Time Comparisons

	Applications					
	Vector Addition		**1D-FFT**		**Matrix Multiply**	
	OpenCL	OCL	OpenCL	OCL	OpenCL	OCL
Execution Time (msec)	28.66	28.66	880.64	880.65	8.21	8.21
Notes	• Vector length 11.5 million 32-bit floats • I-O bound • NVIDIA kernel		• 1024 1K 1D FFTs • 400 GFLOPS ($5*NLog_2(N)$ • I-O bound • Kernel from Apple SDK		• Dimensions A(800 x 1600), B(800 x 800), C(800 x 1600) • I-O Bound (250 GFLOPS) • NVIDIA kernel	

Another set of measurements that were taken were the execution times for these applications. Table 6 lists these measured execution times. Based upon these results, the OCL performance as measured in execution times for the codes written with OpenCL and OCL were, within experimental error, the same.

4.1.2 Event Capture Synchronization across Platforms.

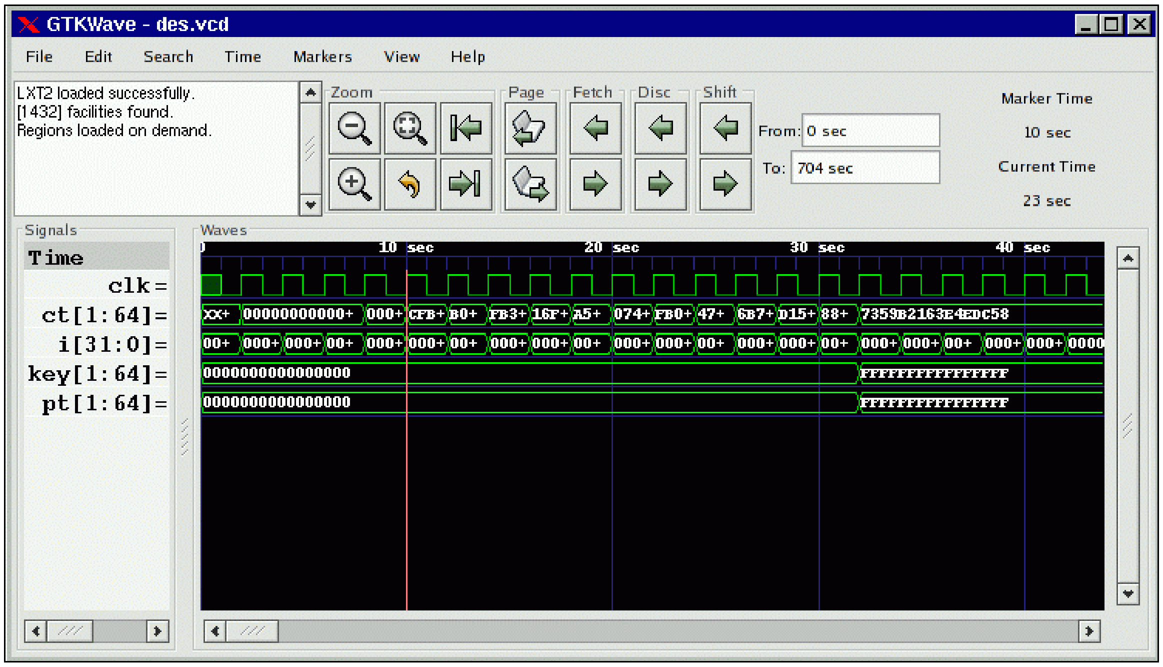
The OpenCPI system was designed for multiple processing platforms including CPUs, GPUs, FPGAs and GPPs. To properly measure system timing, a timing infrastructure provided the timing of the events through synchronization across the complete system. The results of a common timing capability are addressed here.

While an application-level timekeeping clock time required a common clock time, capturing the relative time across a system did not. Time across a system required an understanding of the relationships between the clocks used on the different platforms. Several techniques were experimented with to achieve this.

For FPGA platforms OpenCPI had a built-in measurement mechanism to understand the round-trip time for a GPP to access a FPGA's registers using load and store instructions across the control plane. This was used to accurately capture the difference between the clocks on the GPP and on the FPGA.

On the GPP Linux itself, the CPU's high-speed clock, which executed at GHz, executed independently of the system time obtained via the OS APIs. Here each platform component entity had a local clock. The relative behaviors of the clocks were measured. The local times were timestamped, and during post-processing the respective timestamps were normalized to single timescale.

The post-processing of event data sorted all of the events into a common timeline, and also converted the data to a standard event timing format used by hardware simulators. This was viewed with the open source simulation viewer GTKWave, as shown in the screen shot Figure 21.



```
LXT2 loaded successfully.
[1432] facilities found.
Regions loaded on demand.
```

From: 0 sec
To: 704 sec
Marker Time: 10 sec
Current Time: 23 sec

Figure 21. Viewing Timed Events with the GTKWave

4.2 Application and Component Examples

During the course of the project several components and applications were developed from components, and these were placed in the open source repository as examples for future users. In addition to the OpenCV applications and components that have been described, twelve additional components and three additional applications were developed and tested.

To demonstrate the implementation selection and scoring capabilities, a best-fit application and a set of simple sine wave generation components were developed and tested. To illustrate how an application executed different implementations of the same component specifications, an application was executed that requested different distortion and throughput values, and the infrastructure automatically selected different implementations. When the acceptable distortion increased, a higher throughput implementation was used.

4.2.1 Frequency Shift Key Radio Application.

The final phase of the project was to take an existing DoD application and do a comparison by first executing its original code on its original platform. That code was then to be re-engineered and re-implemented with OpenCPI, and then re-executed on the same platform.

An agreement was made with a government agency to execute the Frequency Shift Key (FSK)/ Radio on its intended platform, which was the N210 platform. The platform-enabling work was completed while waiting for the application to arrive.

The original application was ultimately not available in an unclassified form, and an alternative application was received to be executed on the original N210 platform. However, the supplied unclassified code was incomplete due to proprietary restrictions, and only some of the code was executable. The FSK modem application was a send and receives radio waveform that is described in Figure 22.

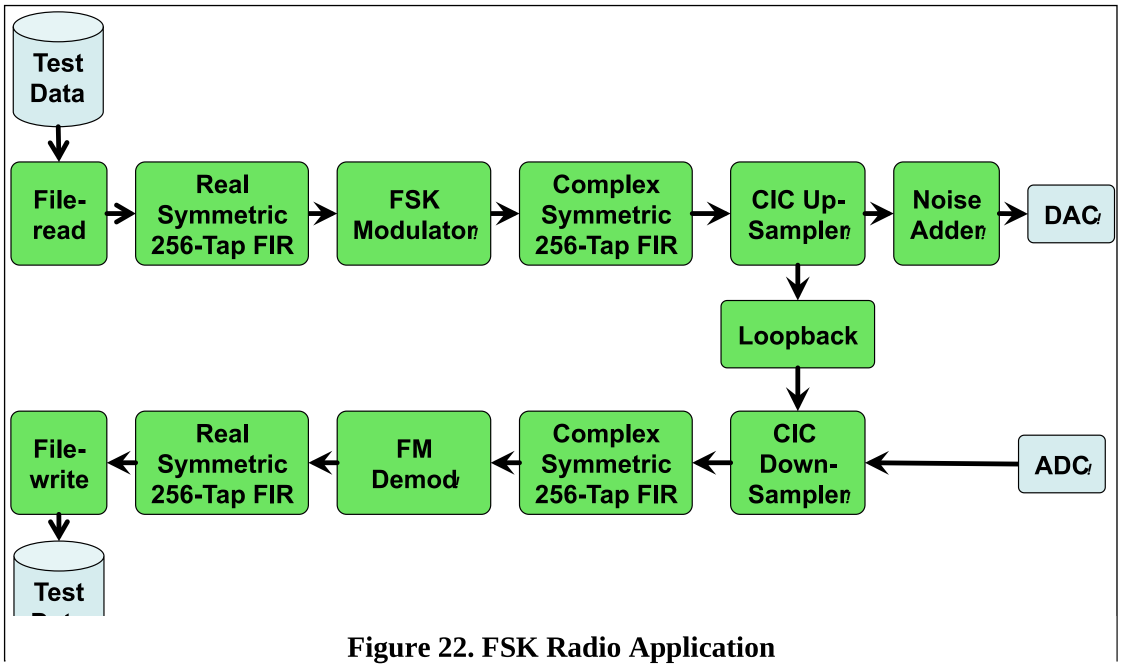

Test Data
File-read
Real Symmetric 256-Tap FIR
FSK Modulator
Complex Symmetric 256-Tap FIR
CIC Up-Sampler
Noise Adder
DAC
Loopback
File-write
Real Symmetric 256-Tap FIR
FM Demod
Complex Symmetric 256-Tap FIR
CIC Down-Sampler
ADC
Test

Figure 22. FSK Radio Application

All of the executable component implementations had been written in software. This configuration represented the functionality of the delivered VHDL code plus the file I-O, which received data from a file for the modulation, and then stored the demodulated received data in another file. The loopback component was used in testing without the actual ADC/DAC hardware. This loopback configuration was executed for testing purposes. In the complete application, which was not received due to proprietary restrictions, there was an additional DSP code that performed the pulse-shaping of the data before the modulation, and then performed the baud tracking. Also, in the real hardware platform, further up/down Intermediate Frequency (IF) and Radio Frequency (RF) conversions were based on other hardware proprietary controls.

To make the best of the situation, the application as received was re-engineered as an OpenCPI application in which the component definitions were created for all of the components, and the software implementations were created for all of the components except for the DSP code. An example application named *fsk_modem* was created that executed all the components with separate software and FPGA implementations. This was executed in a software-only environment. However, since the test data and the test vectors were not received, the software versions of the components were not verified as being numerically correct.

The components that were created for the application were the Symmetric Filter Real, the Symmetric Finite Impulse Response (FIR) Filter Complex, the FSK Modulation, the Cascade Integrator-Comb (CIC) High Pass Filter, the Loopback[36], the Noise Generator[37], the DDS Frequency Generator, the Mixer, the CIC Low Pass Filter, and the Frequency Modulation (FM) Demodulator.

[36] which emulate DAC->ADC

[37] which injected Gaussian noise

The VHDL source code was analyzed as part of the plan to use the HDL Authoring Model FPGA workers to wrap VHDL codes. These workers were based upon the same component specifications that the software implementations were written to. Since all of the original code was VHDL, the HDL Authoring Model for VHDL was enhanced to meet these needs. Compliable VHDL skeletons that were generated for the VHDL components were based on the component definitions created during the re-engineering process.

This application was completed for the execution of all the components, the software implementations, and the VHDL skeletons for the HDL components. The wrapped and functional FPGA components were incomplete.

A number of enhancements depended on third party technology, some of which was not open source and some of which was hardware. In these cases the results would require any user or developer to obtain, install and possibly purchase this other technology. Some of these were also developed and tested against a particular version of third part technology, but there was no further support for keeping it up to date. In these cases updates may be required to exercise the enhancements.

4.2.2 Documentation.

Finally, a number of these tasks were essentially proof-of-concept efforts that established OpenCPI's feasibility and suitability, but require additional efforts to achieve complete functionality and performance. A generic Authoring Model Reference [20] was developed that specifies the concepts of OpenCPI.

The OpenCPI's internal software architecture allowed most major modules to be optionally executed as runtime drivers. A user can configure and use only the modules needed. There are four types of plug-in driver modules, which are the containers, the RDMA/Internal dataplane, the Message/External dataplane, and the library. This project delivered one new container driver, the OCL for GPUs, and one dramatically enhanced container, the HDL for fully discoverable Ethernet-based FPGAs. Also, three new RDMA/Internal Dataplane drivers were developed, the OFED, the RDMA-over-UDP, and the RDMA-over-Ethernet. There were two new Message/External Dataplane drivers, DDS and the one-way CORBA for SCA. A library driver model was created as one of the ease-of-use tasks, and one library driver that executed a typical search path and a directory hierarchy was developed.

Lower level platform infrastructure models were developed on the platforms, including the two newest FPGA platforms, the Altera Stratix4 development PCI-Express board and the Ettus N210 Xilinx/Spartan3A on Ethernet w/ADC/DAC. The hardware-specific modules that make up these OpenCPI platforms are now also OpenCPI. A Linux kernel driver for both low-level PCI-based access/DMA and privileged Ethernet-level protocol processing was fully implemented. It was based on an earlier skeletal prototype.

In addition to the major runtime modules, this project created and enhanced the nine development tools listed Table 7. To provide examples and tests for users, and as a result of the integration experiments, fifteen new applications were developed, along with the thirty components used in them. These are useful not only as application building blocks, but also as regression tests.

Table 7. Development Tools

Name		Purpose
ocpidds	new	Interconverting DDS and OpenCPI metadata
ocpiocl	new	Compiling OpenCL source code to vendor-specific binaries
ocpigen	updates	Generating code skeletons major updates for VHDL and OCL
timeCvt	new	Generating standard VCD files for event/waveform viewing
ocpisca	update	Generating OpenCPI metadata from SCA metadata
scripts	new	Building HDL code for Altera
scripts	new	Building OCL code for OpenCL
scripts	new	Building HDL VHDL code (vs. Verilog).
spreadsheet template	new	Analyzing performance data

This work motivated other opportunistic improvements to all of the aspects of OpenCPI, and these improvements are in hundreds of code files. As a result of this project OpenCPI is appropriate and usable to more users, has more configurations, and addresses more technologies.

A summary of the open source documentation that was created by this and its preceding effort that is available on the OpenCPI website [2] is presented in Table 8.

Table 8. Open Source Documentation

PDF File	Description
OpenCPI_Technical_Summary.pdf	Provides an overview of the OpenCPI technology and architecture along with introductory material common to the other OpenCPI reference documents.
OpenCPI_Generic_Authoring_Model.pdf	Specifies the concept of an OpenCPI authoring model, and defines aspects common to all OpenCPI authoring models.
OpenCPI_CDK_Reference.pdf	Describes the OpenCPI Component Development Kit which is a collection of command line and "make" level tools for developing OpenCPI components (workers).
OpenCPI_Application_Control_Interface.pdf	Describes the C++ interface for launching and controlling OpenCPI application.
OpenCPI_RCC_Reference.pdf	The purpose of this document is to define the OpenCPI RCC Authoring Model.
OpenCPI_OCL_Reference.pdf	The purpose of this document is to define the OpenCPI OCL (OpenCL) Authoring Model.
OpenCPI_HDL_Reference.pdf	The purpose of this document is to define the OpenCPI HDL Authoring Model.
OpenCPI_HDL_App_Workers.pdf	This document describes the HDL application workers in the OpenCPI component library.
OpenCPI_HDL_Device_Workers.pdf	This document describes OpenCPI HDL device workers, which are IP blocks for FPGAs that on the "front" side present standard OpenCPI WIP-profile interfaces for use by applications, and on the "back" side attach to external devices via pins of the FPGA.
OpenCPI_HDL_Infrastructure.pdf	This document describes the OpenCPI infrastructure IP blocks and how they are used as the platform for OpenCPI applications on FPGAs.
OpenCPI_Time_Performance.pdf	Describes the OpenCPI absolute time service and the time and performance instrumentation (TMPI) profiling API.
OpenCPI_RDMA_Protocol.pdf	This document is a functional specification for the OpenCPI data plane RDMA data transfer protocol (OCPIRDT).
OpenCPI_Reference_Platform_Specification.pdf	Describes the OpenCPI Reference Platform in detail. The document provides a parts list for the most recent reference platform along with the software configuration on the system. The document also details the steps needed to bring up the FPGA board with an OpenCPI bitstream.
CP289_v1_0.pdf	Proposed modification to the JTRS SCA specification to support specialized hardware processors.

5.0 CONCLUSIONS

The purpose of OpenCPI was to exploit and manage diverse technologies in embedded systems. This project increased the coverage of embedded systems by supporting technology diversity and component-based development. Some of the key relevant attributes of these types of systems included component based architectures with their associated plug & play, reuse, and integration benefits. These component based systems included well-defined, open, and portable application programming environments for embedded, heterogeneous, and multi-processor applications. Also included were Metadata descriptor formats for the application structure, the deployment, the constraints, the connectivity and the hardware platform elements. A control plane model included a control from within an application through a Human/Computer Interface (HCI) and from a remote control that was outside of the application. A management model was developed for the install/upgrade/uninstall and the startup/shutdown of the application and hardware subsystems. Support from commercial off the shelf design and development tools was also addressed.

Several applications and components were developed as examples. Contributions were made to the Authoring Models for different processing technologies, which added support for GPUs as an OpenCPI processing resource. The platform adaptation and enablement added support for the Altera FPGA platforms and tools. The support of data transport technology allowed for the Optical Fabrics Enterprise Distribution (OFED)/Infiniband usage for inter-component dataflow. The integration and coexistence with other frameworks demonstrated the usage of OpenCPI with the Open Computer Vision (OpenCV) library. The development of the ease-of-use enhancements allowed simple Extensible Markup Language (XML)-based applications to be adopted with component implementation preferences. The development of the application and component libraries supported search paths of the heterogeneous component libraries. The performance measurements supported event synchronization across the independent subsystems.

The underlying OpenCPI baseline architecture was a software-defined platform, originally motivated by the Department of Defense (DoD) Programmable Communication Terminals. It was originally based on and was compliant with the Software Communication Architecture (SCA) developed in the Joint Tactical Radio System (JTRS) program [4].

The specific baseline software platform that was developed had capabilities in several areas critical to the embedded DoD needs, including communications management that was suitable for fabric-based interconnection technologies. The control model did not require the related data plane overhead. The standardized integration of functions on FPGAs and Digital Signal Processors (DSPs) did not require any additional middleware. A standards-based portability model for FPGA applications was developed that was Hardware Description Language (HDL) independent and met the industry standard Open Core Protocol-International Partnership (OCP-IP) specifications. A scalable controlled data plane approach was developed for multi-processing configurations.

In addition to these technical enhancements, several integration and application experiments were performed to assess and demonstrate how OpenCPI could coexist with other relevant frameworks for image and software radio processing, and how well OpenCPI served as an application environment for specific applications. These efforts included instrumentation and measurements.

Since OpenCPI was based on and embraced the principles of open source software, the software, firmware, FPGA codes, tools, and documentation were all posted on the *Opencpi.org* web site and released under an open source license. The key technical results of this work were embodied in the new and improved evolution of OpenCPI, as it addressed a combination of technical gap filling, hardening and ease-of-use enhancements.

Outreach and community building exposed OpenCPI to additional users inside and outside of the defense and intelligence communities. Some of this was based on *ad hoc* meetings and briefings, but there were a number of specific events and workshops where OpenCPI was briefed at conferences. The community building was supported by meetings, test development, and web development, and was highlighted by a workshop at the Wireless Innovation Forum meeting.

This project addressed the key functional gaps in the original open source prototype that inhibited the adoption of OpenCPI. The technology was harden and matured to increase the Technology Readiness Level (TRL). The barriers to OpenCPI development, testing, adoption and implementation were addressed, and the suitability for specific applications was determined. The documentation of this effort was significant, and the feedback was positive. Based on the support of the dataplane drivers, the GPU containers and the FPGA platforms, patterns for supporting new technologies have emerged.

6.0 RECOMMENDATIONS

Based on experience and conclusions of this project, and the fact that interest and experience are growing, it is believed that further efforts and investments are warranted. However, new efforts specifically targeted at OpenCPI should not be research oriented, but rather focused on establishing a solid, perpetual open source community with critical mass to stay up to date with the relevant technologies.

Modest government support is justified to maintain a baseline and provide confidence and credibility. There are numerous examples of government support for baseline governance and maintenance of such a resource. Building and maintaining a repository of components and platform support drivers would be beneficial to the community. Other sponsored projects for particular technologies should be based on specific needs of users, projects and programs.

The specific goals of the application trade study are still very much valid. The bulk of the platform and application re-engineering work was completed, and additional efforts in the platform and VHDL support took the place of the part of the application study work that could not be undertaken. Finishing this work would be value, feasible, and modest, although the proprietary platform baseline is still an issue.

More and more developers assume a GUI-centered development process, and building the appropriate technical bridges to enable the OpenCPI component and application development to take place in the Eclipse environment would improve its acceptance. The government-sponsored RedHawk framework is an example of this, but there are others. Even commercial open source packages such as Code-Sorcery (now part of Mentor Graphics) may be a good integration target. Similarly, building technical integration bridges with MatLab developments, or the open source Octave equivalent, would allow OpenCPI to connect with other aspects of the overall application development process.

Several areas of additional technological development would be valuable, including but not limited to multilevel security, Tilera many-cores, Eclipse Integration, and GPU Direct Memory Access (DMA). The availability of perpetual testing, continuous integration, additional tutorial examples, driver documentation, bug-tracking and road-mapping would be invaluable. A clearer path is needed for new developers to contribute to these and similar technology drivers.

The ease-of-use focus on applications and components did not apply to the driver development, which is still only available to OpenCPI experts. While the internal architecture has evolved to facilitate these drivers, additional guidance and/or documentation needs to be developed.

While there were enthusiastic early users in electronic warfare and signals intelligence, the OpenCPI community is currently finite. The dominant community building focus was to incubate several different embedded communities. A number of useful tools, technologies and practices were identified and tested for testing platforms, components, and applications. Continual institutionalizing, monitoring, and managing OpenCPI are beyond the scope of this project. Now what is needed is a perpetual and comprehensive development and testing community that would keep all of the developed functionality fresh and current.

Further study relating to the acquisition issues for embedded development is warranted. While the application trade study intended to look at a particular application and how it would be improved when using OpenCPI methodology, this does not capture the benefits of component-based design and the open source heterogeneous platform support across multiple projects and programs. The commercial barriers to sharing at this level are significant and costly. An ecosystem of support services that can support multi-vendor heterogeneous platforms is a key ingredient. This is not so much a technical initiative as a policy challenge that deserves further attention.

7.0 REFERENCES

1. Scott, III, J.M., "Open Component Portability Infrastructure (OPENCPI) ," AFRL-RI-RS-TR-2009-257, Mercury Federal Systems, Inc., Arlington, VA, Nov 2009.
2. Kulp, J., "OpenCPI- Open Component Portability Infrastructure," URL: http://opencpi.org/. Accessed Feb 6, 2013.
3. Kulp, J., "OpenCPI Technical Summary," URL: https://github.com/the_Opencpi/the Opencpi/raw/master/doc/OpenCPI_Technical_Summary.pdf. Accessed January 16, 2013.
4. Global Security, "Joint Tactical Radio System Programmable, Modular Communications Systems," URL: http://www.globalsecurity.org/military/systems/ground/jtrs.htm. Accessed February 7, 2013.
5. Siegel, S. and Kulp, J., "OpenCPI HDL Infrastructure Specification," URL: https://github.com/the_Opencpi/theOpencpi/raw/master/doc/OpenCPI_HDL_Infrastructure.pdf. Accessed February 4, 2013.
6. Miller, J. and Kulp, J., "OpenCPI RDMA Protocol Specification," URL: https://github.com/the_Opencpi/theOpencpi/raw/master/doc/ OpenCPI_RDMA_Protocol.pdf. Accessed February 4, 2013.
7. Liu, T.J., "A Real-Time Computer Vision Library for Heterogeneous Processing Environments," MIT, MS Thesis, 2010. URL: http://dspace.mit.edu/bitstream/handle/1721.1/66439/755631660.pdf.
8. Blewitt, A., "Hudson Renames to Jenkins," URL: http://www.infoq.com/news/2011/01/jenkins, January 31, 2011. Accessed February 7, 2013.
9. Kulp, J. and Miller, J., "OpenCPI RCC Authoring Model Reference," URL: https://github.com/the_Opencpi/theOpencpi/raw/master/doc/ OpenCPI_RCC_Reference.pdf. Accessed February 4, 2013.
10. Sridhar, N. and Hallstrom, J.O., "Generating Configurable Containers for Component-Based Software," Proceedings of the 6th Component-Based Software Engineering Workshop at ICSE 2003, Portland, OR, May 3-4, 2003.
11. Ketcham, C., Kulp, J. and Pepe, M. "OpenCPI OCL Authoring Model Reference," URL: https://github.com/the Opencpi/the Opencpi/raw/master/doc/OpenCPI_HDL_Reference.pdf. Accessed February 7, 2013.
12. Kulp, J., "OpenCPI Application Control Interface Specification," ," URL: https://github.com/the Opencpi/the Opencpi/raw/master/doc/OpenCPI_Application_Control_Interface.pdf. Accessed February 4, 2013.
13. Pepe, M. and Kulp, J., "OpenCPI Component Developer Kit (CDK) Reference," ," URL: https://github.com/the Opencpi/the Opencpi/raw/master/doc/ . Accessed February 4, 2013.
14. Open Source Alliance, "OpenFabrics Alliance Software Being Utilized by More than 44 Percent of the TOP500 Systems," URL: http://www.hpcwire.com/hpcwire/2010-11-16/openfabrics_alliance_software_being_utilized_by_more_than_44_percent_of_the_to.500_systems.html. Accessed February 7, 2013.
15. PrismTech, "OpenSplice DDS," URL: http://www.prismtech.com/opensplice. Accessed February 7, 2013.

16. Ettus Research, "USRP N200/N210 Networked Series," URL: https://www.ettus.com/content/files/07495_Ettus_N200-210_DS_Flyer_HR_1.pdf. Accessed February 12, 2013.

17. Pepe, M., and Kulp, J., "OpenCPI Component Developer Kit (CDK) Reference," URL: https://github.com/the Opencpi/the Opencpi/raw/master/doc/The OpenCPI_CDK_Reference.pdf. Accessed February 7, 2013.

18. Kiat, C.L.W., Software Defined Radio Design for an IEEE 802.11a Transceiver using Open Source Software Communications Architecture (SCA) Implementation::Embedded (OSSIE), URL: http://www.amazon.com/Transceiver-Communications-Architecture-Implementation-ebook/dp/B007PU5S28#_ Kindle Edition (2012).

19. Kulp, J. and Siegel, S., "OpenCPI HDL Authoring Model Reference Manual," URL: https://github.com/theOpencpi/theOpencpi/raw/master/doc/OpenCPI_HDL_Device_Workers.pdf. Accessed February 4, 2013.

20. Kulp, J., "OpenCPI Generic Authoring Model Reference," URL: https://github.com/the Opencpi/the Opencpi/raw/master/doc/OpenCPI_Generic_Authoring_Model.pdf. Accessed February 4, 2013.

8.0 LIST OF SYMBOLS, ABBREVATIONS, AND ACRONYMS

ACI	Application Control Interface
ADC	Analog-to-Digital Converter
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BSV	Bluespec System Verilog
CBD	Component-Based Development
CDK	Component Developer’s Kit
CentOS	Community Enterprise Operating System
CF	Core Framework
CIC	Cascade Integrator-Comb
CORBA	Common Object Request Broker Architecture
COTS	Commercial off the Shelf
CPU	Central Processing Unit
CSV	Comma Separated Value (file format)
CUDA	Compute Unified Device Architecture
DAC	Digital to Analog Converter
DDC	Digital Down-Converter
DDS	Data Distribution Service
DG	DataGram
DMA	Direct Memory Access
DoD	Department of Defense
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
FCCM	Field-Programmable Custom Computing Machines
FIFO	First In/First Out
FIR	Finite Impulse Response
FM	Frequency Modulation
FPGA	Field Programmable Gate Array

FSK	Frequency Shift Keying
FST	Fast Simulator Trace
GBE	GigaBit Ethernet
GFLOPS	10^9 FLoating-point Operations per Second
GIOP	General Inter-ORB Protocol
GPGPU	General Purpose Graphics Processing Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HCI	Human/Computer Interface
HDL	Hardware Description Language
I	In-phase
IDL	Interface Description Language
IF	Intermediate Frequency
IO	Input-Output
IP	Internet Protocol
ISO	International Organization for Standardization
iWARP	Internet Wide Area RDMA Protocol
JTRS	Joint Tactical Radio System
L2	Link layer
LGPL	Lesser GNU Public License
LXT	interLaced eXtensible Trace
MILS	Multiple Independent Levels of Security
MIT	Massachusetts Institute of Technology
MPI	Message Passing Interface
NFS	Network File System
NIC	Network Interface Card
OCL	OpenCL Authoring Model
OCP-IP	Open Core Protocol — International Partnership

OFED	Open Fabrics Enterprise Distribution
omniORB	a robust high performance CORBA ORB for C++ and Python
OpenCL	Open Computing Language
OpenCPI	Open source Component Portability Infrastructure
OpenCV	Open Computer Vision
OpenGL	Open Graphics Library
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
OSSIE	Open source Software Communication Architecture Implementation: Embedded
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
PPS	Passive Parallel Synchronous
Q	Quadrature
RCC	Resource-Constrained C-language
RDMA	Remote Direct Memory Access
RF	Radio Frequency
RHEL	Red Hat Enterprise Linux
RoCCE	RDMA over Converged Enhanced Ethernet
SCA	Software Communications Architecture
SCARI	Software Communications Architecture Reference Implementation
SDR	Software Defined Radio
SIGINT	Signals Intelligence
SLOC	Source Lines of Code
Soft-RoCEE	Soft-RDMA over the Converged Enhanced Ethernet
SWAP	Size, Weight, and Power
TCP	Transmission Control Protocol
TRL	Technology Readiness Level

UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Universal Resource Locator
UUID	Universal Unique Identifier
VCD	Value Change Dump (file format)
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VZT	Verilog/VHDL Zipped Trace
WCI	Worker Control Interface
WIP	Worker Interface Profile
WSI	Worker Streaming Interface
XML	Extensible Markup Language